令和元年度 特別研究報告書

非同期システムコール処理機構の提案

龍谷大学 理工学部 情報メディア学科

学籍番号 ： T160395
氏名 ： 中島 健

指導教員 ： 芝 公仁 助教，三好 力 教授

内容梗概

プロセスがデバイスやカーネルの機能へアクセスするためにはシステムコールを発行する必要がある．システムコールの発行にはプロセスからカーネルへの処理の移行により，速度の低下が発生する．また，プロセスはシステムコールを一度発行すると処理が完了するまで他の処理を行うことができないため，処理に時間がかかるシステムコールを発行すると処理速度への影響が発生する．

本論文ではシステムコールを発行するスレッドと処理するスレッドを分け，非同期でシステムコールを処理できる機構を提案する．また，提案機構を複数のスレッドで動作させる評価実験を行う．実験の結果，システムコールを複数スレッドで処理したときの処理速度が向上した．

目 次

1　はじめに

ユーザプロセスがデバイスやOSの機能へアクセスするためにはシステムコールを発行する必要がある．システムコールはカーネル内での処理をプロセスの代わりに行うインタフェースであり，これを用いることでカーネル内での意図しない動作を防ぎ，セキュリティとOSの安定性を向上させる．一方でシステムコールの発行はCPUの処理をプロセスからカーネルに移行する．このとき，それまでの処理のキャッシュのクリアが発生し，速度が低下する．また，システムコールを発行すると速度の低下が考えられる．本論文では非同期システムコールについて述べる．本機構はシステムコールを処理するためのスレッドを作成し，プロセスのスレッドと分けることでシステムコールを非同期で処理する．これによってプロセスはシステムコールの処理を待つことなく処理が行える．

以下，本論文では，3章で提案機構の構成を述べ，4章で提案機構の動作について述べる．また，5章で評価を行い，提案システムの有用性を示す．

2　関連研究

システムコールのオーバヘッドを削減するための関連研究に FlexSC: Flexible System Call Scheduling with Exception-Less System Call [1] がある．FlexSC ではシステムコールのオーバヘッドにシステムコールの同期実装やユーザモードからカーネルモードへ遷移するときの時間やその際に発生するキャッシュの再配置などシステムコールによって引き起こされるオーバヘッドについて挙げられている．これを解決するために例外のないシステムコールによる柔軟なシステムコールのスケジューリングが提案されている．例外とはシステムコール発行時のユーザモードとカーネルモードの切り替えのことであり，例外のないシステムコールではユーザ空間とカーネル空間の共有メモリを作成し，その中に syscall page とよばれるシステムコール発行に必要な情報を保持している．この syscall page を介してシステムコール発行と処理を行い，システムコールによるオーバヘッドを低減している．この FlexSC によって Apatch や MySQL などのパフォーマンスが向上している．

3　async_syscall の構成

async_syscall は図 1 に示すようにカーネル内に async_syscall を作成した．async_syscall は専用の処理スレッドがシステムコール非同期処理を行うための機構である．処理スレッドはプロセスから発行されたシステムコールを確認し，システムコール関数を呼び出し，実行結果をプロセスに返却する．従来のシステムコール発行ではプロセスがシステムコールを発行するとシステムコールの結果を待つ必要があったが，専用のスレッドが本機構を通してシステムコールを処理することで，プロセスはシステムコールの結果を待たずに処理を続けることができるようになる．

3.1　システムコールリクエスト

async_syscall ではプロセスが設定したリクエストに応じたシステムコールを発行する．リクエストは syscall_info 構造体で必要な情報を保持している．syscall_info には以下の情報がある．

- state: リクエストの状態
- num: システムコール番号
- arg[6]: システムコール引数

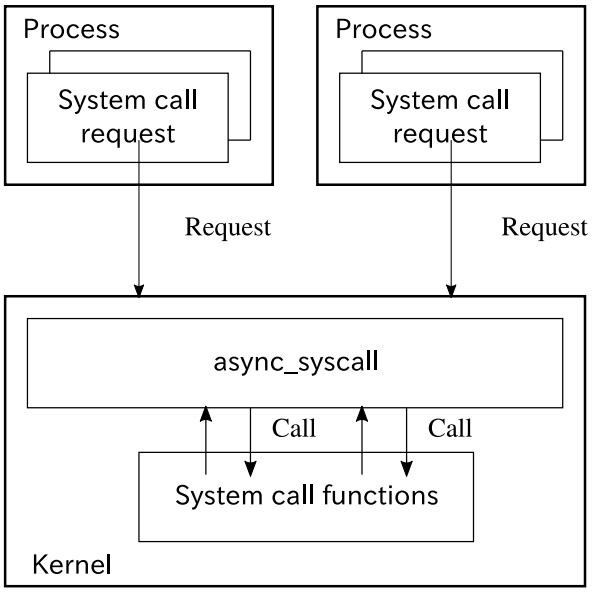

図 1　async_syscall を用いたシステムコールの発行

- ret: リクエストの実行結果

num にはリクエストのシステムコール番号が格納されている．arg にはシステムコール引数が格納される．システムコールは 0 個から 6 個までの引数を取るため，arg に長さ 6 の配列を確保する．ret にはシステムコールリクエストの実行結果が格納される．state にはリクエストの処理状況が格納されている．また state には以下の四つの状態がある．

- NONE: リクエストが入っていない状態
- HASCALL: 処理されていないリクエストが存在する状態
- EXECUTING: 処理中
- COMPLETED: リクエストの処理が完了し，結果を保持している状態

state は初期化されると NONE となり，要求を追加されると HASCALL に変化する．HASCALL が処理をするときだけ EXECUTING に変化する．処理が終了すると COMPLETED になり，結果が回収されると NONE に変更される．

3.2 リクエスト領域

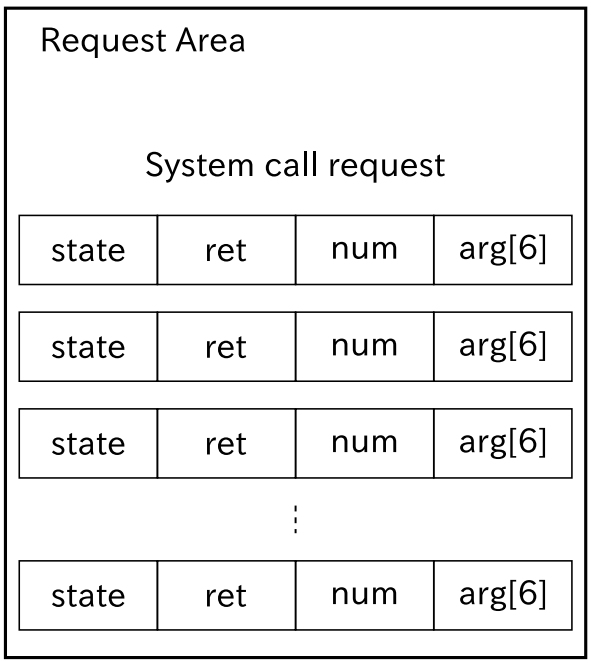

図 2　リクエスト領域の構成

システムコールリクエストはユーザ空間とカーネル空間の双方から参照される．ユーザ空間とカーネル空間ではデータの移行にシステムコールを用いるが，システムコールの発行数を削減するためにプロセスにカーネルから参照と書き込みが行えるメモリ領域，リクエスト領域を作成する．これを利用してシステムコールの回数を減らしつつ，syscall_info をカーネルから参照することができ，システムコール発行時のオーバヘッドを削減することができる．リクエスト領域のサイズはリクエストの数によって変化する．syscall_info 型は 72 バイトの長さであり，要求数が増えるほど syscall_info の数が増えるため，サイズも大きくなる．また，サイズは 4096 の倍数になる．

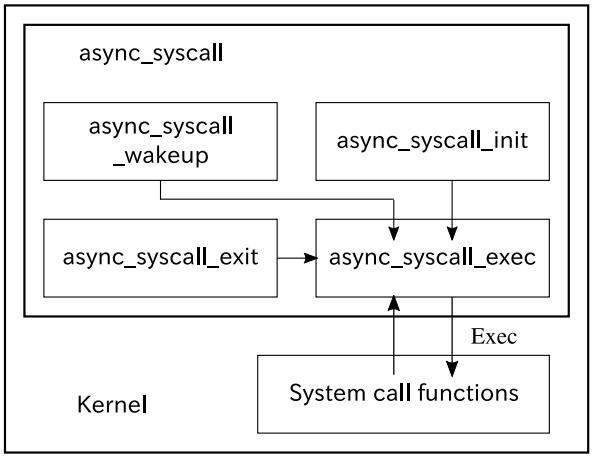

図 3　async_syscall の構成

3.3　システムコール

async_syscall は以下の 4 つのシステムコールによって構成されている．

- async_syscall_init
- async_syscall_exec
- async_syscall_wakeup
- async_syscall_exit

async_syscall_init は request_thread_queue や exec_thread_queue，排他制御に必要な mutex などカーネル内で使用するデータを初期化する．引数は取らず，返り値は常に 0 を返す．async_syscall_exec はリクエストを処理する．引数にはリクエスト領域の先頭のアドレスと一括で発行したいシステムコールの個数を取る．返り値は state の状態が NONE，HASCALL，EXECUTING，COMPLETED 以外である場合は-1 を返し，async_syscall_exit により正常に終了した場合は 1 を返す．async_syscall_wakeup は現在 wait している exec_thread の処理を再開させる．また，リクエストの state を確認して COMPLETED が一つも存在しなければ request_thread を wait させる．引数には，リクエスト領域の先頭アドレスとシステムコールの一括発行数を取り，exec_thread が再開した場合は 1 を返し，それ以外の場合は 0 を返す．async_syscall_exit は exec_thread を終了させるシステムコールである．exec_thread が終了するためのフラグを true にした後，wait している exec_thread を再開させる．引数には何も取らず，結果は常にに 0 を返す．

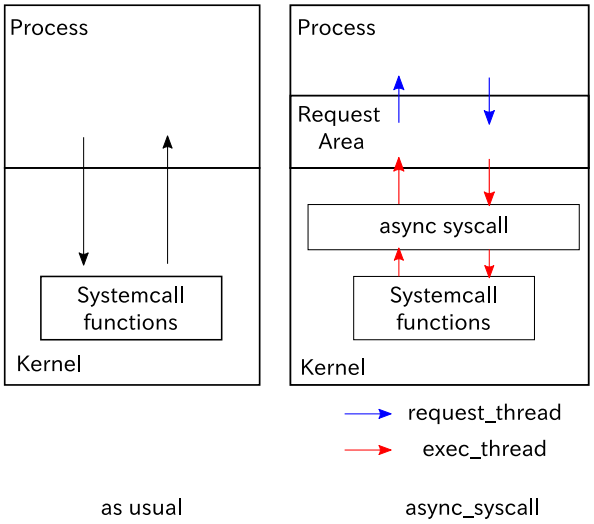

図 4　従来の発行時のスレッドの比較

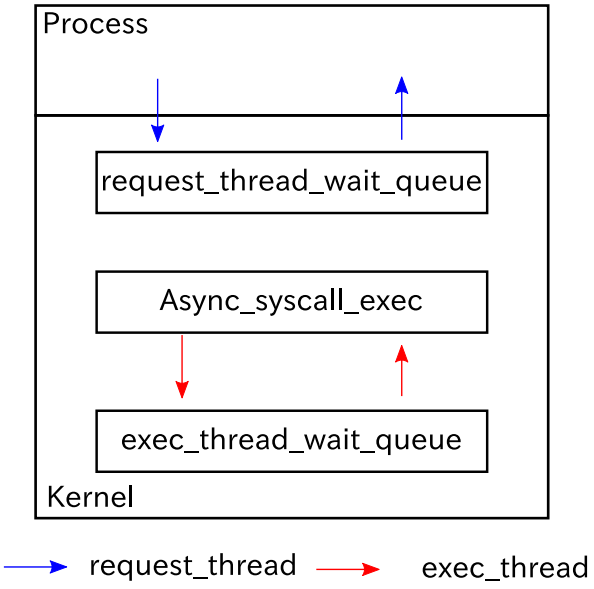

図 5　それぞれのスレッドと queue

3.4　スレッド

図 4 に示すように従来のシステムコールでは同じスレッドがシステムコールの発行，処理を行う．しかし，発行と処理を同じスレッドで行うとシステムコールの処理が完了するまで他の処理を行うことができない．

async_syscall では図 5 に示すようにシステムコールリクエストの作成と実行結果の回収を行うスレッド request_thread とシステムコールリクエストを処理するスレッ

ド exec_thread がある．exec_thread は一つ以上の任意の数を指定することができる．exec_thread はシステムコールの処理を行うスレッドであるため，exec_thread の数を増やすことでシステムコールを並列で処理することが可能となる．そして，リクエストの発行と実行を異なるスレッドが行うことでシステムコールの非同期処理を可能にする．また，request_thread と exec_thread は wait することがあるため，それらを待たせるための queue，request_thread_wait_queue と exec_thread_wait_queue を用意し，待機状態になった request_thread と exec_thread を待機させる．async_syscall_exec や async_syscall_wakeup，async_syscall_exit は wait はこの queue に入って wait しているスレッドの処理をすべて再開させる．

4　async_syscall の動作

4.1　全体の動作

async_syscall の動作を図 6 に示す．図中の青矢印は request_thread を表し，赤矢印は exec_thread を表す．まず最初に request_thread はリクエストを発行する前にリクエスト領域を作成する．リクエスト領域の作成が完了すると request_thread はシステムコールリクエストを作成する．次に exec_thread が state を確認する．確認中は state が変更されることを防ぐため，すべて確認し終えるまでロックをかける state が HASCALL であれば state を EXECUTING に変更し，システムコールリクエスト処理を行う．処理が完了すると処理結果を ret に格納し，state を COMPLETED に変える．request_thread は state を確認し，COMPLETED であった場合は結果を取得し，state を NONE に変える．

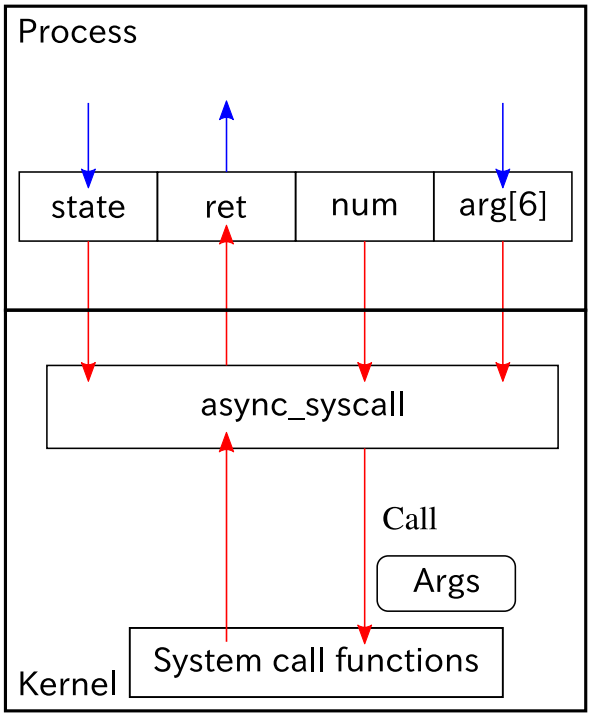

図 6　async_syscall の動作

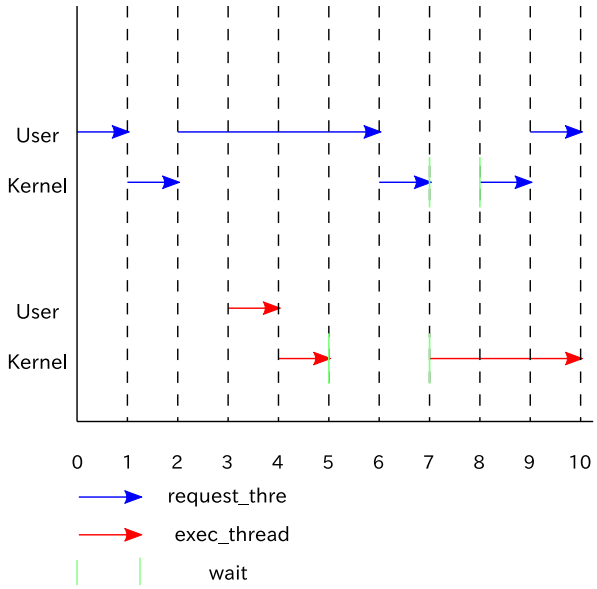

図 7　request_thread と exec_thread のユーザとカーネル遷移

4.2　リクエストの作成

プロセスは発行したいシステムコールのリクエストを async_syscall から参照できるようにするため，カーネルと共有するメモリ領域を確保する．メモリの確保が完了すると async_syscall_init を発行し，カーネル内の async_syscall の実行に必要な情報を初期化する．その後，プロセスは exec_thread を任意の個数生成する．exec_thread の作成後，プロセスはリクエストを作成する．

request_thread がリクエスト領域を作成してからリクエストの実行結果を取得するまでの request_thread と exec_thread のユーザ，カーネル間の遷移を図 7 に示す．時刻 1 では request_thread がユーザとカーネルの共有するメモリ領域を作成する．時刻 2 では async_syscall_init を実行するためにユーザからカーネルへ遷移する．時刻 3 では request_thread が exec_thread を生成する．時刻 4 では生成された exec_thread がシステムコール async_syscall_exec でカーネルへ遷移する．時刻 5 では request_thread がリクエストを生成するまで exec_thread が wait する．時刻 6 では request_thread がリクエストを作成する．時刻 7 ではリクエストの作成を終えた request_thread が exec_thread を再開させるために async_syscall_wakeup を実行する．async_syscall_wakeup を実行した request_thread は，exec_thread がリクエストの処理を一つ完了させるまで wait する．async_syscall_wakeup により処理を再開した exec_thread はシステムコールリクエ

ストの処理を実行する．時刻 8 では exec_thread がシステムコールリクエストの処理を行い，が一つでも完了すると request_thread を再開させる．時刻 9 では request_thread がユーザへ戻り exec_thread はシステムコールリクエストの処理を続ける．時刻 10 では request_thread がリクエストの実行結果を確認する．request_thread はリクエストを作成すると一度 wait するが，exec_thread が一度でも処理を行うと request_thread が再開される．このため，exec_thread がシステムコールリクエストを処理している間に request_thread が処理の結果を待つことなく別の処理を非同期に実行することができるようになる．

5　評価

async_syscall の基本性能を評価するために従来のシステムコールとの処理時間の比較と exec_thread の数を変更したときの計測の二つの実験を行った．

5.1　オーバヘッドの確認

async_syscall と従来のシステムコール発行のオーバヘッドを確認するための評価実験を行った．実験は表 1 の環境で行った．また，リクエスト領域のサイズはリクエストの数によって変動する．システムコールのオーバヘッドの評価を行うために従来のシステムコールと今回提案した async_syscall でそれぞれ gettid を連続で発行した．実験では発行回数を 0 回から 400 回まで 20 回ずつ増加させ，システムコールの処理が完了するまでの時間を時間を計測した．評価結果を図 8 に示す．従来の連続発行と async_syscall の一括発行も回数を増やすたびに処理時間も増加した．また，発行回数が 360 回以上では従来のシステムコール連続発行による処理時間が sync_syscall より長くなった．約 360 個のリクエストを処理するまで従来の発行方法よりも async_syscall の処理時間が

表 1　評価環境

カーネル	GNU/Linux 4.19.91
CPU	Intel Corei5-9600K 3.70GHz
ライブラリ	glibc 2.28-10

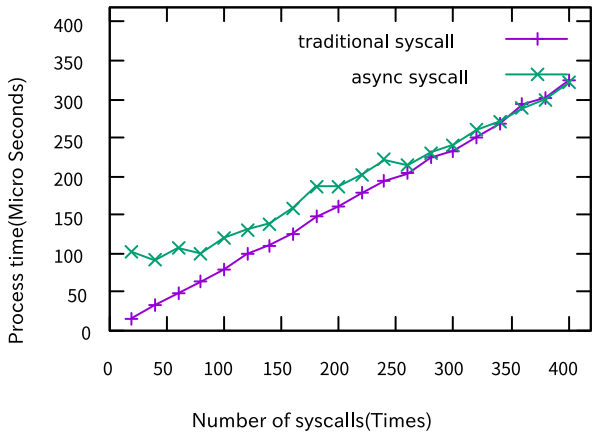

図 8　従来のシステムコール連続発行と async_syscall による一括発行の比較

表 2　実験内容

システムコール	read
一括発行数	100 回
読み込むファイルサイズ	10MB
一度に読み出すサイズ	100KB

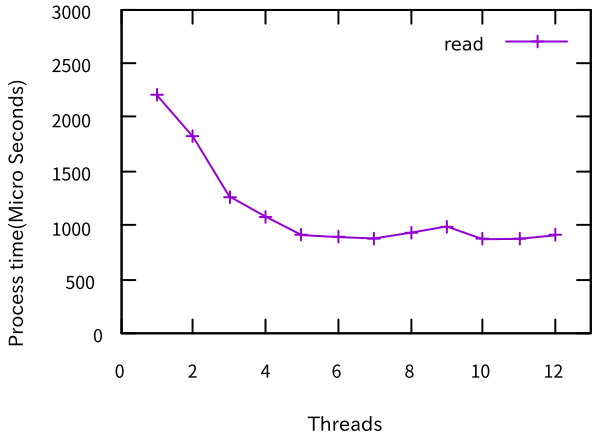

図 9　exec_thread の個数による処理速度の比較

長かった原因は，async_syscall でシステムコールを処理する時間がシステムコールリクエストのオーバヘッドよりシステムコールリクエスト実行するときのオーバヘッドに時間がかかったことだと考えられる．約 360 回までのシステムコールの発行には従来の連続発行が，それよりも多くシステムコールを発行する場合は async_syscall によって発行することでより短い時間でシステムコールを処理することができる．

5.2　マルチスレッドによる処理

exec_thread によるシステムコールの並列処理を確認するためにスレッド数を変化させたときの処理時間を計測した．実験は表**??**で示すように 10MB のファイルを read する時間を計測する．read システムコールは 100 回発行させ，1 回あたり 100KB を読み込む．なお，実験にはディスクの IO が発生しない tmpfs 上のファイルを使用した．実験の結果を図 9 に示す．exec_thread を増やすことで処理時間は短くなった．exec_thread が 7 個生成されたときに処理時間が最短になり，シングルスレッドに比べて約 60%短い．exec_thread が増えると処理時間が短縮されるが，スレッド数が増えると処理時間

の変化がゆるやかになり，6 個前後では，大きな変化が見られなくなった．これは実験に使用した Intel Core-i5 9600K は CPU のコアが個つであり，スレッド数も 6 個であったため，6 個以上のスレッドでは処理時間の変化が見られなかったと考えられる．

6　おわりに

本稿では非同期システムコール処理機構について述べた．本機構ではシステムコールの処理を専用のスレッドである exec_thread が行うことにより，システムコールの要求と処理を非同期処理で行う．また，プロセス，カーネル間からのシステムコールリクエストへのアクセスはプロセス，カーネルが共有するリクエスト領域を予め作成して用いることでシステムコール発行時のオーバヘッドの低減させる．評価実験では exec_thread の数を増やし，並列処理を行うことで処理時間の短縮を確認した．今後はサーバや IO 処理といったシステムコールが数多く発行される状況や負荷の高いシステムコールの処理に適用することを今後の課題とする．

謝辞

本研究を行うにあたり，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] Soares, L. and Stumm, M.: FlexSC: Flexible System Call Scheduling with Exception-Less System Calls., *OSDI* (Arpaci-Dusseau, R. H. and Chen, B., eds.), USENIX Association, pp. 33–46 (2010).