Vol.2015-OS-134 No.14
2015/8/5

Exception-lessシステムコールを用いた高速デバイスに対する効率的I/O処理

中島 基陽[1,a)]　追川 修一[2]

概要：ハードディスク，Flash SSD など，現在使われている二次記憶は低速であり，このようなデバイスにアクセスするにあたってはデバイスの処理時間中に他の処理を行うことのできる割込処理が適切である．しかし近年，DRAM とほぼ変わらないレイテンシを持つメモリ素子の登場により，適切な I/O 処理手法が変化しつつある．このような背景の中，高速なデバイスに対するより良い I/O 処理手法の確立は一つの課題である．本研究では，システムコールの発行にソフトウェア割り込みが不要である exception-less システムコールを用いた高速デバイスに対するより効率的な I/O 処理手法を提案，実装し，その評価を行う．

キーワード：割込処理，ポーリング，ディスク I/O，高速デバイス

1. はじめに

ハードディスクや，NAND フラッシュを用いた Flash SSD など，今現在広く使われている二次記憶は低速である．従って，それらのデバイスの応答をビジーウェイトで単に待つのは非効率であり，その為にそれらのデバイスの入出力にあたって，デバイスの処理完了までの間に他の仕事を行う為に割込処理が用いられる．一方，近年 Persistent Memory と呼ばれる高速な次世代不揮発メモリ素子の実用化が期待されている．Persistent Memory ベースのストレージは超低レイテンシであり，この入出力にあたっては割込処理よりもビジーウェイトでポーリングする方が効率的であることが分かっている [1]．このようにデバイスが高速化する中，デバイスに対する効率的な処理方式が変化してきており，高速なデバイスに対応したより良い処理手法の確立は一つの課題となっている．したがって本研究では，高速なデバイスに対するより効率的な入出力処理手法を提案，実装し，その評価を行う．

2. 割り込みとそのオーバーヘッド

割込処理は低速なデバイスに対して有用である．OS はこの場合デバイスに対して入出力する為のデバイスコマンドを割り込み設定で発行し，他の処理に移る．デバイスの処理が完了すると，デバイスは CPU に割り込みを掛け，OS は対応した割込処理を行う．デバイスが低速であれば，デバイスの処理完了までの間に CPU が他の処理を行った方が CPU 資源を有効利用できるので効率的である．この場合にデバイスの処理終了をポーリングして定期的にチェックすると，ポーリングのオーバーヘッドが大きくなってしまう．一方，例えば Persistent Memory ベースの超低レイテンシストレージに対しては割込処理を行うよりもビジーウェイトでポーリングする方が効率的である [1]．これは，ビジーウェイトによるポーリングでデバイスの処理完了を待つ方式に比べて割込処理の方がオーバーヘッドがかかることに起因する．割込処理におけるオーバーヘッドの要因はいくつかありそのひとつは，割り込みが CPU にかかった時にと命令パイプラインのフラッシュが発生してしまうことである．命令パイプラインはアーキテクチャにもよるが，大量の投機的実行を含む最適化が行われている場合があり，命令パイプラインのフラッシュは大きなコストとなりうる．また，デバイス処理が終了するまでの間に CPU が他の処理を行った場合，キャッシュ・TLB が汚染される．特に，他のユーザプロセスの処理に移る場合はページテーブルの切り替えが発生し，異なるメモリ空間での大きなキャッシュ・TLB 汚染が発生する．アーキテクチャによってはページテーブルの切り替えに伴って TLB をフラッシュしてしまう場合もある．また，高速なデバイスに対しては，OS 側が割り込み処理の為にする仕事が，本来不要な余計な仕事となってしまう．これは OS 側の割込処理のソフトウェアオーバーヘッドとなる．また，割込処理を行う場合には，割り込みしない場合に比べてデバイ

[1] 筑波大学システム情報工学研究科コンピュータサイエンス専攻
University of Tsukuba, Tsukuba, Ibaraki 3050005, Japan
[2] 筑波大学システム情報系情報工学域
[a)] s1420677@u.tsukuba.ac.jp

ス側にも割り込みのハードウェアオーバーヘッドがかかってしまう．高速なデバイスではこのオーバーヘッドは無視できない大きさである [1]．更に，割込処理の中で，他にする仕事が無い場合に CPU を省電力状態にする場合がある．その場合には CPU の省電力状態からの復帰コストもかかってくる．

従来の低速なデバイスに対しては，このような割込処理の特徴によるオーバーヘッドよりも割込処理による性能向上の方が上回る．しかし，高速なデバイスに対してはこれらの複数の要因がポーリング処理に対して割込処理の性能を落としてしまうことになる．したがって，高速デバイスに対してはこのような割込処理のオーバーヘッドと如何に付き合っていくかが問題である．

3. FlexSC と exception-less システムコール

本節では，本研究において用いられている FlexSC の exception-less システムコール [2] について説明する．[2] では，システムコール処理を非同期でバッチして行い，システムコールの発行にあたってトラップを要さない exception-less システムコールを用いた処理手法が提案されている．この手法によってシステムのソフトウェア割り込みの回数を削減することができる．システムコールを発行する場合，通常はソフトウェア割り込みを要する．ユーザ空間からシステムコールを発行するにあたって，システムコールのユーザ空間での処理を行い，トラップを行う為の専用命令を用いて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を行う．このトラップ命令により，CPU はユーザモードからカーネルモードへと移行し，より高い特権レベルへと遷移する．その後，カーネル内でのシステムコールの処理に移り，システムコールの処理が終わると，カーネル空間からユーザ空間へのコンテキストスイッチを行ってユーザ空間の処理に戻る．これが一般的なシステムコールの一連の流れとなる．しかし，トラップは命令パイプラインのフラッシュを必要とし，前述のように命令パイプラインのフラッシュコストは無視できないものである．また，カーネル内での処理によってキャッシュ・TLB が汚染されてしまうという問題もある．

そこで，[2] では，exception-less システムコールを導入している．Exception-less システムコールの発行にはトラップを必要としない．Exception-less システムコールの発行はユーザ・カーネル間の共有ページにシステムコール要求を書き込むことによって行われ，その後ユーザプロセスはユーザ空間内の処理を続行することができる．Exception-less システムコールによって発行されたシステムコールの処理はカーネルスレッドが行う．カーネルスレッドは共有ページから書き込まれたシステムコール要求を読み取り，それに応じて処理を行う．そしてその共有ページの同じエントリに結果も書き込み，ユーザプロセスはそれを読んで結果を受け取るというものである．これにより，命令パイプラインをフラッシュしてしまう回数を削減することができる．更に，ユーザプロセスは可能な限りそのユーザ空間における処理を行うことができ，従来のシステムコールにおける，カーネル空間の処理によるキャッシュ汚染を防ぐことができる．

この手法はシステムコールをバッチして処理を行うことで性能を向上させている．また，非同期にシステムコールを処理するが，いずれはユーザプロセスにおいてシステムコールの結果を待ち，同期する必要がある．この FlexSC に依存した効率的なユーザプログラムを書くこともできるが，既存のプログラムを FlexSC を exception-less システムコールを効率的に利用したものに書き換えるのはコストがかかる．そこで，FlexSC-Threads というユーザレベルスレッドパッケージも考案されている [2]．この FlexSC-Threads を用いると，従来の同期的なシステムコールを用いたプログラムを exception-less システムコールを使用したものにすることが可能である．ただし，FlexSC はシステムコールをバッチして処理を行うことで性能を向上させているので，スレッド数が少ない場合には見込める性能向上が小さいと予想される．

4. 提案手法

本研究では，FlexSC の exception-less システムコールを用いた高速デバイスに対するより効率的な I/O システムコール処理手法を提案する．高速なデバイスへのアクセスには割込処理を用いるよりもポーリングを行った方が効率的であるということが分かっている [1]．これは，割込処理にはソフトウェア・ハードウェア両面でのオーバーヘッドがかかることが原因である．そのオーバーヘッドを受け入れるよりは，ビジーウェイトでポーリングしていた方が効率的だということである．しかし，デバイスの処理が完了するまでの間ビジーウェイトでポーリングするのではなく，その間に別のカーネルの処理を行った方が効率的である可能性がある．そこで，本研究における提案手法では，デバイスの処理が完了するまでの間に FlexSC における exception-less システムコールのカーネル内での処理を行う．

[1] では，高速なデバイスに対して割込処理を行うよりもポーリングを行った方が好ましい理由として，(1) 割り込みオーバーヘッド，(2) 他の仕事をすることによるキャッシュ・TLB 汚染，(3)CPU の省電力状態からの起床コストが挙げられている．(1) 割り込みオーバーヘッド，(3)CPU の省電力状態からの起床コストに関しては，提案手法では I/O においてポーリングを行うので割り込みオーバーヘッドがかからず，デバイスコマンドを発行した後，仕事が無い場合に省電力モードに移行することもない．(2)

他の仕事をすることによるキャッシュ・TLB 汚染に関しては，特に，デバイスの処理が完了するまでの間に別のユーザプロセスの処理を実行しようとすると，ページテーブルの切り替えが発生して場合によっては TLB のフラッシュが発生し，異なるメモリ空間における処理によってキャッシュ・TLB が汚染される．しかし，exception-less システムコールを処理する為のカーネルスレッドの処理ならば，そのままカーネル空間内の処理を継続できるのでページテーブルを切り替える必要が無く，この時に TLB のフラッシュが発生することもない．更に，システムコールの呼び出しが exception-less システムコールによって行われていた場合，FlexSC においては exception-less システムコールを処理するカーネルスレッドを実行する CPU の仕事は exception-less システムコールの処理が優先されるので，キャッシュ汚染が抑制できる．つまり，本提案手法では，高速なデバイスに対して割込処理のオーバーヘッドをかけることなく，より小さいキャッシュ・TLB 汚染で，デバイス処理中の処理が可能となる．

5. 設計

本節では提案手法の実装について述べる．提案手法では，デバイスに対してポーリングを行う合間の仕事として exception-less システムコールの処理を行う．Exception-less システムコールの発行にあたっては，ユーザプロセスがユーザ・カーネル間の共有ページである syscall ページに対してシステムコール要求の書き込みを行う．ユーザプロセスは FlexSC の場合と同様に，exception-less システムコールを発行した後は場合によってユーザレベルスレッドを使い，そのプロセス内の他の処理を行うものとする．そして，syscall ページから取得できる exception-less システムコールの結果が存在せず，exception-less システムコールの結果待ちで他にする仕事が無い場合にユーザプロセスは専用のシステムコールで exception-less システムコールの処理完了を待ってブロックするものとする．Syscall ページに書き込まれたシステムコール要求はそれを処理するカーネルスレッドである syscall スレッドによって処理される．Syscall スレッドを実行する CPU は固定し，プロセッサにアフィニティを与える．つまり，syscall スレッドを実行しない CPU は exception-less システムコール処理にあたってアクセスする命令やデータによってキャッシュ汚染されることがなく，exception-less システムコール処理によってアクセスする命令やデータを特定の CPU キャッシュに集中させることで，より良いキャッシュヒット率が得られる．

Exception-less システムコールを処理する syscall スレッドは，syscall ページに存在するシステムコールエントリの数だけ生成される．ディスクに対して I/O を行う必要がある場合にはポーリングを行う．具体的には，該当する syscall スレッドでデバイスコマンドを発行し，別の syscall スレッドの処理に移る．そして，複数の syscall スレッドの処理を実行する合間に，デバイスの状態チェックし，ポーリングを行う．デバイスの処理が完了していたら，対応したカーネルスレッドの処理へと移り，入出力処理，syscall ページへの結果の書き込みを行う．これにより，高速なデバイスに対して割込処理のオーバーヘッドをかけることなく，より小さいキャッシュ・TLB 汚染で，デバイス処理中の仕事が可能となる．

6. 実装

今回，MIT が開発した 32 ビット OS である xv6[3] 上で提案手法を実装した．この上に FlexSC-Threads 相当の exception-less システムコール発行可能なユーザレベルスレッド，syscall スレッド相当の exception-less システムコール処理部分，超低レイテンシデバイスのシミュレータとそのデバイスドライバを加えた．Exception-less システムコールを発行するプロセスは exception-based システムコールである flexsc_register() システムコールにより syscall ページの確保，初期化やマップ，syscall スレッド相当のデータ領域の確保と初期化などを行い，更にユーザ空間で FlexSC-Threads 中で用いるミューテックス，セマフォや条件変数領域の確保，初期化，syscall ページチェック用スレッドの fork などを行う．Exception-less システムコール発行の際には syscall エントリへシステムコール要求を書き込み，他のユーザレベルスレッドへと処理を移す．Syscall ページをチェックするユーザレベルスレッドが処理の完了したエントリを見つけた場合にそのスレッドが起こされ，その時点で処理できることがなくなった場合には他のプロセスに処理を譲る．Xv6 にはカーネルスレッドが存在しないので，syscall スレッドの処理へはスケジューラからの分岐から入る．Syscall スレッドの処理に入っていない時に syscall ページへの書き込みを検知した場合に syscall スレッドの処理へと移る．上述のように xv6 にはカーネルスレッドが存在しないので，現在の実装では単一の固定の CPU が順々に syscall ページに提出された exception-less システムコール要求を処理していく．Exception-less システムコールの処理中にブロックする場合もあり，その場合には syscall ページにその状態を書き込んで次の exception-less システムコールの処理へと移る．本実装では syscall ページの 1 エントリは 32 ビットであり，1 ページの 4KB 中には 128 エントリ入るものとしている．また，超低レイテンシデバイスのシミュレータとして，特殊な CPU を一つ動かす．超低レイテンシデバイスをシミュレートするにあたって，実際は RAM ディスクにアクセスするものとする．この CPU は割り込み禁止状態で常に超低レイテンシデバイスへのアクセスを監視し，アクセスがあった場合にはキャッシュ無効化領域へ一回アクセスしてから，割り込み設定でのアクセスならば割り込み

処理の為の CPU 間割り込みを起こし，そうでなければ超低レイテンシデバイス側の処理が完了したフラグを立てるというものである．これを処理するデバイスドライバは，通常ブロックが不要な RAM ディスクへのアクセスを従来のディスクの様にブロックを用いるものとして扱い，その処理完了を超低レイテンシデバイスのシミュレータである CPU によって示してもらい，必要ならば処理完了通知である割り込みをしてもらうということをしている．これにより，DRAM アクセス程度のレイテンシの高速デバイスをシミュレートしている．

7. 評価

実験環境としては QEMU 上で実行を行い icount=0 としている．実験方法としては，シミュレートした高速デバイスに対して exception-based システムコールを用いた場合と exception-less システムコールを用いた場合で以下のようなプログラムをそれぞれ実行し，比較を行った．

(1) u 個のユーザレベルスレッドを生成し，メインスレッドが各スレッドの処理終了を待つ．
(2) 各スレッドが異なるファイルを開く．
(3) 各スレッドが write() システムコールによる 512 バイトのそれぞれのファイルへの書き込みを d 回繰り返す．
(4) 各スレッドが各ファイルを閉じる．
(5) 再び同じファイルを各スレッドが開く．
(6) 各スレッドが read() システムコールによる 512 バイトのそれぞれのファイルからの読み込みを d 回繰り返す．
(7) 各スレッドが各ファイルを閉じる．

実装した提案手法を評価した結果が以下の図 1 である．図 1 では，$d = 8$ としてユーザレベルスレッドの個数を変化させて exception-based システムコールを用いた場合と exception-less システムコールを用いた場合を比較した場合の結果である．ユーザレベルスレッド数が 8 までは exception-based システムコールを用いた場合でも exception-less システムコールを用いた場合でもほぼ変わらない結果となっているが，それ以降から差が開き，exception-less システムコールを用いた場合の方が実行時間が少なくなっている．Exception-based システムコールを用いた場合にはユーザレベルスレッドの個数はそのまま負荷となるので負荷の上昇に伴って実行時間も増えている．Exception-less システムコールを用いた場合には syscall スレッドの処理が並行して行われる為にユーザプロセス側の処理が軽減され，負荷に対する処理時間の伸びが緩やかになっていると考えられる．この点で，高速デバイスに対して exception-less システムコールを用いた場合の性能向上の可能性が見られる．

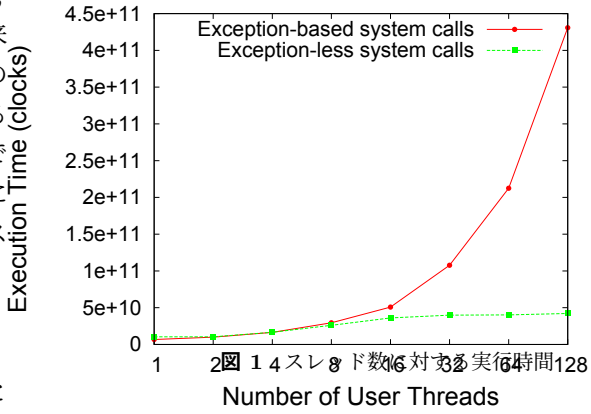

図 1 4スレッド数に対する実行時間

8. 関連研究

本研究では，exception-less システムコールによってソフトウェア割り込みの回数を削減し，更に，ポーリングによってハードウェア割り込みも避けるという手法を提案した．このように，本研究における提案手法ではソフトウェア割り込み，ハードウェア割り込みの両方に伴うオーバーヘッドを抑え，更なる性能向上を狙っている．また，提案手法の中では CPU にアフィニティを与えたり，別メモリ空間での処理を避けることでキャッシュ・TLB 汚染を軽減し，ポーリング中に行う処理によるキャッシュ・TLB 汚染の問題を軽減させている．これらについて，この節ではハードウェア割り込み，ソフトウェア割り込みのオーバーヘッド，キャッシュ効率向上の為のアプローチに関する本研究に関連する研究について述べる．

8.1 ハードウェア割り込み

[4] では，クロック割り込みによるシステムノイズが性能に悪影響を与えることを示し，スマートタイマと言うクロック割り込みの代替手法を提示している．このことからも，デバイスによるハードウェア割り込みによるシステムの性能が無視できないことが分かる．また同様に，割り込みのオーバーヘッドに関する研究として，[1] がある．この研究では，DRAM と同程度の速度を持つ次世代 NVM ベースのストレージを想定し，このような高速なデバイスに対する I/O 処理手法として割り込みとポーリングのそれぞれを行った結果の性能比較を行っている．これは，I/O 処理手法としての割り込みがどれほど性能に影響を与えるかという定量的な研究でもあり，結果としては，Persistent Memory とも呼ばれる次世代の高速なメモリ素子をベースとしたストレージに対しては割り込み処理を行うよりもポーリングを行ってデバイスの処理完了をビジーウェイトで待つ手法の方が効率的だということが分かっている．これらの研究から，特にデバイスによるハードウェア割り込みがシステムの性能に与える悪影響と，それを避けることによる性能向上が見込めることが分かる．

8.2 ソフトウェア割り込み

一方，ソフトウェア割り込み・従来のシステムコールのオーバーヘッドについての定量的な研究，exception-less システムコールを用いることによる性能向上を示すものとして [2] がある．この研究では，従来の発行にソフトウェア割り込みを要する exception-based システムコールのオーバーヘッドを定量的に述べており，そのオーバーヘッド要因である命令パイプラインフラッシュ，別メモリ空間内の処理による大きなキャッシュ・TLB 汚染の考察も行っている．これにより，従来のシステムコール呼び出しに伴うオーバーヘッドが無視できないことが分かり，本提案手法のようなアプローチが有用であることが分かる．

8.3 キャッシュ効率

提案手法では CPU にアフィニティを与え，別メモリ空間での処理を避けることでキャッシュ・TLB 汚染を軽減し，ポーリング中に行う処理によるキャッシュ・TLB 汚染の問題を軽減させている．CPU にアフィニティを与える別アプローチの研究としては，[5] がある．これは，VM のハイパーバイザとゲスト OS の処理をそれぞれ異なる CPU 上で行い，それぞれにプロセッサアフィニティを持たせる．ハイパーバイザとゲスト OS が別の CPU 上で実行されるとすると，ハイパーバイザを実行する CPU のキャッシュにはハイパーバイザの処理内容の命令，データが集中し，キャッシュヒット率も向上する．ゲスト OS を処理する CPU のキャッシュもまた同様である．本提案手法では exception-less システムコールを処理する CPU がアフィニティを持ち，それらの CPU のキャッシュの内容がカーネル内のシステムコール処理に集中してキャッシュヒット率を向上させている点で似ているものとなっている．

また，[6] では Mach 3.0 において実装された Mach Continuation が提案されている．スレッドがブロックする場合には，その時にカーネルスタックに各レジスタの値を積んでカーネルスタックを保存し，再びそのスレッドが実行される時にそのカーネルスタックからレジスタを復元し，プログラムの実行状態を復元するのが一般的である．しかし，Mach Continuation では，これに加えてもう一つの実行状態復元手法を持つ．それは，スレッドがブロックするにあたってコンテキストを次回起動時に実行する関数，continuation として保存する方法である．スレッドが continuation を用いる方法でブロックされた場合にはスレッドはカーネルスタックを持つ必要が無くなり，スレッド管理の処理に関する時間を削減することが可能である．また，これは使用するカーネルスタックの数が減ることに繋がり，キャッシュミスや TLB ミス率の減少，キャッシュ効率の向上に繋がる．これらの研究で用いられている概念を本提案手法と組み合わせて使うことで更なる性能向上が見込める可能性がある．

9. まとめと今後の課題

今回は高速なデバイスに対して，命令パイプラインフラッシュ，キャッシュ汚染などの割込処理のオーバーヘッドをかけることなく，より小さいキャッシュ・TLB 汚染でデバイス処理中の仕事を可能とする手法を提案，設計，実装を行い，評価を行った．今後の課題としては，より正確な評価を行う為に実験環境を実機に移すこと，より公平な評価を行う為に複数プロセスを fork する実験プログラムでも評価を行うことである．

参考文献

[1] Jisoo Yang, Dave B Minturn, and Frank Hady. When poll is better than interrupt. In *FAST*, page 3, 2012.

[2] Livio Soares and Michael Stumm. Flexsc: Flexible system call scheduling with exception-less system calls. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, pages 1–8. USENIX Association, 2010.

[3] Xv6, a simple unix-like teaching operating system. http://pdos.csail.mit.edu/6.828/2014/xv6.html.

[4] Dan Tsafrir, Yoav Etsion, Dror G Feitelson, and Scott Kirkpatrick. System noise, os clock ticks, and fine-grained parallel applications. In *Proceedings of the 19th annual international conference on Supercomputing*, pages 303–312. ACM, 2005.

[5] Alex Landau, Muli Ben-Yehuda, and Abel Gordon. Splitx: Split guest/hypervisor execution on multi-core. In *WIOV*, 2011.

[6] Richard P Draves, Brian N Bershad, Richard F Rashid, and Randall W Dean. Using continuations to implement thread management and communication in operating systems. In *ACM SIGOPS Operating Systems Review*, volume 25, pages 122–136. ACM, 1991.