低優先度処理を指定可能なリアルタイム処理向けI/Oスケジューラ

高村 成道[1]　鵜川 始陽[1]　岩崎 英哉[1]

概要：近年，リアルタイム性を必要とし，停止できない Web サービスが増加している．それらのメンテナンス処理はサービスを継続した状態で行う必要がある．このようなメンテナンスをオンラインメンテナンスと呼ぶ．オンラインメンテナンスの例には，データの集計や削除がある．オンラインメンテナンスを行う上で，メンテナンス処理の I/O 負荷が Web サービスに悪影響を及ぼすことが問題となっている．たとえば，データベースサーバ上の数百 GB のログファイルを削除するメンテナンスをオンラインで行う場合，大量の I/O 処理が行われデータベース処理が長時間停止するという事態が生じる．このような問題を解決するために，本研究ではメンテナンスの I/O 処理の実行をゆっくりと行う機構を提案する．Linux においては，I/O 処理の実行順序は I/O スケジューラが決定する．そこで，リアルタイム性を必要とする Web サービスのバックエンドで用いられている I/O スケジューラに，低優先度処理を指定する機能を追加した．実験用のデータベースサーバ上で本機構を評価したところ，既存の I/O スケジューラではデータベースと関係のない I/O 処理の実行中にデータベースのスループットが 30% 以下に低下したのに対し，提案する I/O スケジューラでは 75% 以上となった．

A Deadline I/O Scheduler Equipped with Low Priority Assignment

Narimichi Takamura[1]　Tomoharu Ugawa[1]　Hideya Iwasaki[1]

Abstract: The availablity of web services has been incresingly demanded. To achieve high availablity, database servers used in the backends of web services have to respond in a low latency. Maintenance operations (e.g., deleting log files and mining data) executed by an operator may raise a pressure on I/O subsystem, which increases the latency of database processing. To solve this problem, it is desired to execute maintenance operations in low priority. However, the deadline I/O scheduler, which is recommended for database servers on Linux cannot deal with priorities for I/O operations. In this research, we have extended the deadline I/O scheduler to enable us to assign low priority to a process, which is inherited to the children. Our I/O scheduler can suppress the increase of latency by executing the maintenance operations at low priority. We evaluated the usefulness of our I/O scheduler on a database server. While in the deadline I/O scheduler, maintenance operations degraded the throughput of database processing down to less than 30%, in our I/O scheduler they degraded that down to at most 75% by assigning low priority to maintenance operations.

1. 背景

近年，リアルタイム性を必要とし，停止できない Web サービスが増加している．それらのメンテナンス処理はサービスを継続した状態で行う必要がある．このようなメンテナンスをオンラインメンテナンスと呼ぶ．たとえば，データの集計や削除がオンラインメンテナンスで行われる．オンラインメンテナンスを行う上で，メンテナンス処理の I/O 負荷が Web サービスに悪影響を及ぼすことが問題となっている．たとえば，データベースサーバ上の数百 GB のログファイルを削除するメンテナンスをオンラインで行う場合，大量の I/O 処理が行われ，データベース処理が長時間停止するという事態が生じる．

メンテナンス処理の影響を最小限に抑えるための対策として，メンテナンスの I/O 処理を低優先度で行うことが挙

[1] 電気通信大学
The University of Electro-Communications

© 2014 Information Processing Society of Japan

げられる．I/O 処理の優先度指定は，I/O スケジューラに依存する．リアルタイム性が求められるサーバでは，リアルタイム処理向きの I/O スケジューラを用いるのが一般的だが，Linux には，低優先度の I/O 処理の指定が可能であり，なおかつリアルタイム処理向きである I/O スケジューラは存在しない．そのため，リアルタイム処理とオンラインメンテナンスの低優先度指定は両立しない．

そこで本研究では，オンラインメンテナンスによるサービスの停止を防ぐ機構を Linux に実現することを目的とする．そのために，メンテナンスの I/O 処理の実行をゆっくりと行うようなリアルタイム処理向けの I/O スケジューラを提案し（4 章），実装する（5 章）．具体的には，既存のリアルタイム向け I/O スケジューラに対して，低優先度を指定する機能を追加することで実現する．また，実装した I/O スケジューラと既存の I/O スケジューラに対して，I/O 処理のベンチマークによる性能を比較することで評価を行う（6 章）．

2. 関連研究

Linux カーネルにおける I/O スケジューラに関しては，様々な研究が行われている．それらの研究の多くは，I/O 処理の性能改善を目的としている．SSD 向け I/O スケジューラ [1] や Fusion-io のような高速デバイス向け I/O スケジューラ [2] は，各デバイス向けに最適化した新しい I/O スケジューラを開発することで，より高速な I/O 処理を実現するものである．Seetharami ら [3] と Ramon ら [4] は，システムのワークロードを分析し，オンラインで最適な I/O スケジューラに切り替えることで，効率的な I/O 処理を行う方法を提案している．これらの研究は，いずれも低優先度 I/O 処理を考慮していない．

Carl [5] は，帯域制御機能を追加する方法を提案している．この研究では，帯域幅を指定するための優先度クラスを新たに定義することで，帯域制御が可能な I/O スケジューラを実装している．この I/O スケジューラは，任意のプロセスに対して I/O 帯域幅を制御することができるため，低優先度 の I/O 処理を指定することが可能である．しかし，CFQ I/O スケジューラを拡張しているため，リアルタイム処理向けではない．一方，本機構では，リアルタイム処理向けである Deadline I/O スケジューラに対して，特定のプロセスの I/O 処理のスループットを抑える機構を提案した．また，本機構でも各種パラメタを適切に設定することで，低優先度クラスのスループットの調節を行うことが可能である．

バージョン 2.6.24 以降の Linux カーネルでは，cgroups（control groups）[6] という機能で．プロセスグループのリソース（CPU，メモリ，I/O など）の利用を制限・隔離することができる．cgroups を用いれば，特定のプロセスに対して I/O 処理のスループットを Bytes/sec で指定することができる．そのため，スループットを低めに指定することで低優先度処理を実現することが可能である．この機能は I/O スケジューラに依存しないが，利用するにはプロセスごとに設定が必要であり煩雑である．また，低いスループットを指定したプロセスは，他のプロセスによって I/O 処理が行われていない場合も，指定したスループットで処理を行うため，マシンのリソースを活かしきることができない．本研究で提案する I/O スケジューラは，`ionice` コマンドを用いることで，容易に低優先度指定が可能である．また，通常の I/O 処理がない場合に，低優先度を指定したプロセスはマシンのリソースを有効活用して処理を行う．

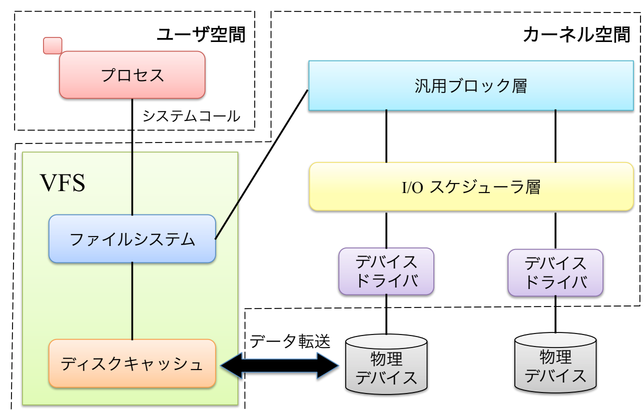


図 1　Linux におけるファイル I/O の概略

3. I/O スケジューラ

I/O 処理は，ユーザ空間にあるプロセスからシステムコールを介して，カーネル空間にあるサービスルーチンを呼び出すことで行われる．カーネル空間では，階層的に処理を行い，最終的に物理デバイスにアクセスする．Linux におけるファイル I/O は，図 1 に示す通り，様々なカーネル要素を組み合わせることで実現されている．I/O スケジューラとは，カーネルに組み込まれたソフトウェアのことで，汎用ブロック層と物理デバイスの中間で動作している．

既存の I/O スケジューラの代表として CFQ と Deadline が挙げられる．優先度指定が可能である CFQ I/O スケジューラは，複数のプロセスが公平に I/O 処理時間を分けあうアルゴリズムであるため，リアルタイム処理向きではない．一方，リアルタイム処理向けである Deadline I/O スケジューラでは，優先度を指定することができないため，オンラインメンテナンス時に本来のサービスに悪影響を及ぼす可能性がある．

I/O スケジューラは，効率的に I/O を処理するために，I/O 要求の並び替えや併合を行う．要求の管理はキューを用いて行う．キューの使用方法は，I/O スケジューリングアルゴリズムによって異なる．I/O スケジューラの概略を図 2 に示す．I/O スケジューラ層は，以下の 2 種類の構造

© 2014 Information Processing Society of Japan

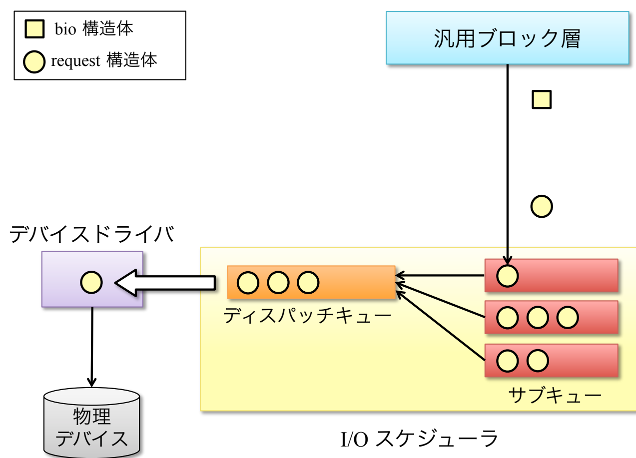


図 2 I/O スケジューラの動作概要

体を利用することで，前後の層である汎用ブロック層やデバイスドライバとの間で要求の転送を行う．

bio 構造体

汎用ブロック層と I/O スケジューラ層の間で利用されるデータ構造である．bio 構造体は，物理デバイスに対するデータの読み書きを依頼するリクエストデータを表す．転送するデータ量，アクセスする領域の最初のセクタ番号やデータ転送の方向（Read or Write）をメンバに持つ．

request 構造体

I/O スケジューラ層とデバイスドライバの間で利用されるデータ構造である．request 構造体は，1 つ以上の bio 構造体から構成される．I/O 要求の実体がこの構造体である．

I/O スケジューラは 2 種類のキューを保持している．

1 つ目は**サブキュー**と呼ばれるもので，スケジューリングアルゴリズムに従って I/O 要求を並び替えるキューである．汎用ブロック層から受け取った I/O 要求は，一時的にこのキューに格納される．格納された I/O 要求は，適切に並び替えられ，もう一つのキューである**ディスパッチキュー**に移動される．キューの数や I/O 要求の並び替え方は，I/O スケジューラによって異なる．I/O 要求をサブキューからディスパッチキューへ移動する動作のことを**ディスパッチ**と呼ぶ．また，ディスパッチを行う関数を**ディスパッチ関数**と呼ぶ．ディスパッチ関数は，I/O 要求の追加に伴って呼び出される．

ディスパッチキューは，request 構造体が I/O 処理を行う順序で格納されるキューである．このキューに格納された request 構造体の処理には，キューに溜まった request 構造体を処理するモード（unplug）と，実際のデータ転送は行わずキューに request 構造体を溜めていくモード（plug）の 2 つがある．通常時は plug となっており，request 構造体がある程度溜まると unplug となる．この仕組みにより，ディスクヘッドの移動を減らすことができる．

I/O スケジューラとその前後の層における動作の流れを以下に示す．

(1) 汎用ブロック層で bio 構造体を作成する．その際，物理的な位置が近い bio 構造体同士を併合する．

(2) 汎用ブロック層で作られた bio 構造体をもとに request 構造体を作成する．受け取った bio 構造体と既存の request 構造体が保持するセクタ番号が近い場合は，新たに request 構造体は作成せず，それらを併合する．

(3) 新しく request 構造体を作成した場合は，それを I/O スケジューラに追加する．多くの場合，受け取った request 構造体を一旦サブキューに格納し，適切に並び替えた後で，ディスパッチキューに移動する．

(4) ディスパッチキューに request 構造体が少ない場合は，plug モードを維持したまま，物理デバイスにデータ転送を行うことなく処理が終了する．ディスパッチキューにある程度の request 構造体が溜まっている場合は，unplug モードになる．

(5) unplug モードの場合，対応するデバイスドライバはディスパッチキューの I/O リクエストを選択し，物理デバイスに対してデータ転送を行う．デバイスドライバによるこの一連の処理を**ストラテジルーチン**と呼ぶ．

4. 設計

4.1 方針

低優先度を指定可能なリアルタイム処理向けの I/O スケジューラを実現するために，既存の Deadline I/O スケジューラを拡張する．拡張の方針は以下の通りとする．

低優先度処理のスループット制限

低優先度の I/O 処理によって本来優先すべき I/O 処理を遅延させないために，通常の I/O 要求が存在する場合は，低優先度の I/O 処理のスループットを抑える．

低優先度処理によるスループットの有効活用

物理デバイスの性能を最大限に引き出すために，低優先度の I/O 要求のみがサブキューに存在する場合は，低優先度であってもスループットを制限せずに処理する．

処理性能の維持

低優先度のリクエストがない場合は，既存の Deadline I/O スケジューラと同等の処理性能を維持する．

ユーザは，ionice コマンドを用いて，優先度指定を行うも

© 2014 Information Processing Society of Japan

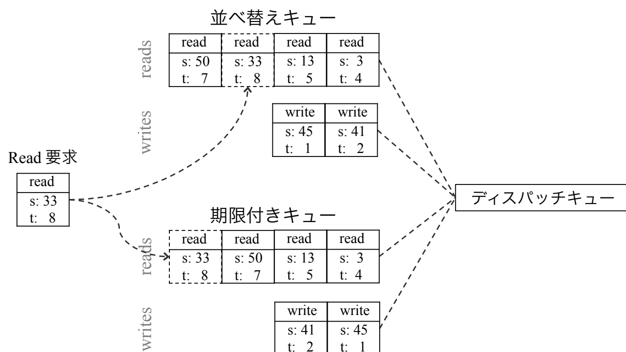


図 3 Deadline I/O スケジューラにおける既存キュー

のとする．この指定の方法は CFQ スケジューラを利用する場合にも用いられる一般的な方法である．なお，`ionice` コマンドは非同期書き込みの場合は優先度指定ができない．そのため，本機構は非同期書き込みではなく，読み出しと同期書き込み（`O_DIRECT` など）を対象とする．

4.2 既存の Deadline I/O スケジューラ

既存の Deadline I/O スケジューラの戦略は以下の通りである．

- 基本的にセクタ番号順に処理する．
- 一定時間処理されなかった要求は優先的に処理する．

この戦略は，図 3 に示す通り並べ替えキューと期限付きキューという 2 種類のサブキューを用いて実装されている．それぞれキューは，Read と Write を区別して利用するため，サブキューは合計で 4 つある．並べ替えキューはセクタ番号を保持し，期限付きキューは I/O 要求がキューに追加された時の `jiffies` の値を保持している．ここで `jiffies` とは，システム起動時からの経過時間を保持するグローバル変数である．Deadline I/O スケジューラは期限付きキューを参照して，一定時間処理されていない要求がないかどうかを調べる．もしそのような要求があれば，その要求を優先的に処理する．Read 要求が追加された場合のキューの例を図 3 に示す．図 3 において，右側がキューの先頭である．また，s は セクタ番号を保持し， t は I/O 要求追加時の `jiffies` の値である．図 3 では，`jiffies` が 8 のときにセクタ番号が 33 である Read 要求が，I/O スケジューラに追加されたときのキューの動作を示している．要求を受け取ると 2 種類のキューに対してそれぞれ追加する．並べ替えキューではセクタ番号順になるように追加が行われる．期限付きキューではキューの末尾に追加が行われる．2 種類のキューに入れられた同じ要求は相互に対応がとられており，片方が削除されるともう片方も削除される．

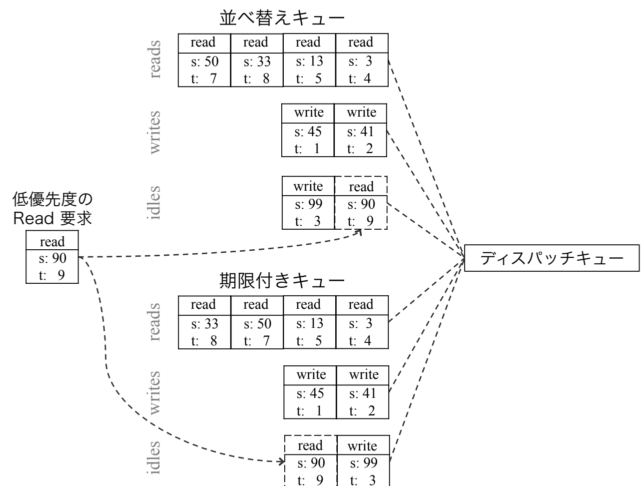


図 4 拡張後の Deadline I/O スケジューラ

4.3 低優先度キュー付き Deadline I/O スケジューラ

本研究では，低優先度の I/O 処理を区別して処理するために，既存の並べ替えキューと期限付きキューに対して，1 つずつキューを追加した低優先度キュー付き Deadline I/O スケジューラを提案する．低優先度の Read 要求が追加された時のキューの例を図 4 に示す．図 4 の idles のキューが本機構で新たに追加するキューである．このキューを低優先度キューと呼ぶ．図 4 では，`jiffies` が 9 のときにセクタ番号が 90 である低優先度の Read 要求が，I/O スケジューラに追加されたときのキューの動作を示している．

低優先度の I/O 要求は，新規に追加した並べ替えキューと期限付きキューにそれぞれ追加される．通常の I/O 要求と低優先度の I/O 要求を区別してキューに配置することで，それぞれの I/O 要求が互いに影響を及ぼさないようにする．

4.4 低優先度の I/O 要求のディスパッチ

本機構は，I/O スケジューラのディスパッチにおいて低優先度の I/O 処理のスループットを制限する．ディスパッチの仕組みは以下の通りである．

（1） 通常の I/O 要求がキューに存在する場合，一定時間間隔で少しずつ低優先度の I/O 要求をディスパッチする．

（2） 通常の I/O 要求がキュー存在しない状態が一定時間継続した場合，スループットを制限せずに低優先度の I/O 要求のディスパッチする．

（1）は，通常の I/O 要求を優先的に処理するための仕組みである．この仕組みを実現するためには，通常の I/O 要求の存在を確かめる必要がある．通常の I/O 要求の確認は，既存のキューに I/O 要求が存在するかどうかを確認することで行う．低優先度の I/O 要求は，通常の I/O 要求が存在する場合は保留され，存在しない場合はディスパッ

© 2014 Information Processing Society of Japan

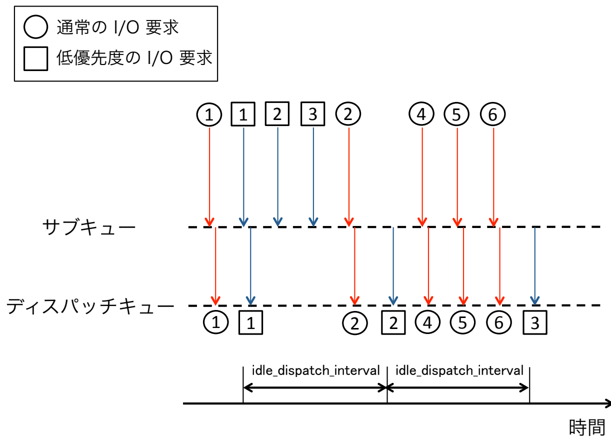


図 5　低優先度の I/O 要求のディスパッチ間隔

チされる．たとえば，図 4 における低優先度の Read 要求がディスパッチされるには，既存のキューに格納されている 6 つの要求がすべてディスパッチされるまで待機する必要がある．

（2）は，低優先度の I/O 処理のスループットを制限するための仕組みである．既存の Deadline スケジューラでは，ディスパッチを行うと，サブキューに存在するすべての I/O 要求はディスパッチキューに移動される．（1）の仕組みだけの場合，通常の I/O 要求のディスパッチが完了した後，すぐに低優先度の I/O 要求のディスパッチが行われる．そのため，短い周期でディスパッチが行われる場合には低優先度キューに要求があまり保留されない．これでは，次に通常の I/O 処理を行おうとした場合．ストラテジルーチンが忙しくてすぐに処理されない恐れがある．そこで，本機構では意図的に低優先度の I/O 要求のディスパッチを遅延させる仕組みを追加する．この仕組みは，一定の時間間隔を空けて低優先度の I/O 要求をディスパッチすることで実現する．これより，低優先度の I/O 要求がゆっくり処理されるようになるため，スループットを抑えることができるようになる．低優先度の I/O 要求を処理する時間間隔を `idle_dispatch_interval` と定義する．この仕組みを導入した場合のディスパッチの動作を図 5 に示す．各 I/O 要求から伸びた矢印と各キューの点線との交点は，キューに I/O 要求が追加された時刻を表す．横軸は時間を表しており，要求を表す図形の数字は要求の到着順序を表す．図 5 において，各 I/O 要求がサブキューに追加された後，通常の I/O 要求はすぐにディスパッチされているのに対し，低優先度の I/O 要求は，一定間隔待機した後にディスパッチされている．なお，`idle_dispatch_interval` は，sysfs という仮想ファイルシステムを通してユーザが設定可能なパラメタとする．

（3）は，低優先度の I/O 処理の際に物理デバイスのスループットを有効活用するための仕組みである．この仕組みを実現するために，2 つのモードを定義する．1 つは，通常の処理を行う「通常モード」，もう 1 つは低優先度の I/O 処理を積極的に行う「バッチモード」である．I/O スケジューラがバッチモードへ移行した場合，（2）で導入したスループット制御は解除される．これにより，低優先度の I/O 処理が制限を受けないスループットで処理されることになる．バッチ処理の最中に通常の I/O 要求が発行された場合は，（1）の仕組みによって通常の I/O 要求が優先的にディスパッチが行われる．この時，通常モードに移行する．バッチモードへの移行は，通常の I/O 要求が一定時間発行されない場合に行われる．この一定時間を `batch_mode_delay` と定義する．それぞれのモード移行の動作を図 6 に示す．`batch_mode_delay` の値は，慎重に定義する必要がある．この時間が短すぎる場合は I/O 要求がディスパッチされる度にバッチモードへ移行するため，低優先度の I/O 処理のスループットを抑えられなくなってしまう．一方，長すぎる場合はバッチモードに移行しなくなってしまうため，スループットを有効活用することができなくなってしまう．`batch_mode_delay` も，sysfs という仮想ファイルシステムを通してユーザが設定可能なパラメタとする．

5. 実装

5.1 低優先度キューの定義

Deadline I/O スケジューラでは，並べ替えキューは `rb_root` 構造体，期限付きキューは `list_head` 構造体を用いて定義されている．これらのキューは Read と Write 用に 2 つずつ定義されている．本研究では，低優先度処理を行うためのサブキューを 1 つずつ定義した．

5.2 ディスパッチの制御

4.4 節で述べたディスパッチ制御に関する 3 つの仕組みを実装した．それぞれの仕組みの詳細を以下に示す．

（1）の仕組みである，通常の I/O 要求がサブキューに存在しない場合にのみ，低優先度の I/O 要求のディスパッチを行う擬似コードを図 7 に示す．ここで，`dd` は Deadline

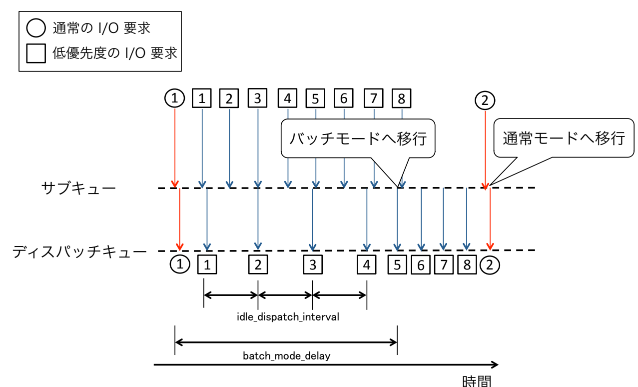


図 6　モード移行の動作

© 2014 Information Processing Society of Japan

```
deadline_dispatch_requests(struct
    deadline_data *dd) {
  reads = !list_empty(dd->sort_list[READ]);
  writes = !list_empty(dd->sort_list[WRITE]);
  lowprios =
    !list_empty(dd->sort_list[LOWPR]);
  struct request *rq;

  // 優先順序はRead > Write > Idle
  if (reads) {
    data_dir = READ;
    goto dispatch;
  } else if (writes) {
    data_dir = WRITE;
    goto dispatch;
  } else if (lowprios) {
    data_dir = IDLE;
    goto dispatch;
  }
  // 3つの並べ替えキューが空の場合は終了
  return;

dispatch:
  rq = dd->sort_list[data_dir];
  deadline_move_request(rq);
}
```

図 7 通常の I/O 要求を優先的に処理する擬似コード

I/O スケジューラの固有のデータであり, `sort_list[]` は, Read, Write, 低優先度（LOWPR）並べ替えキューの配列である．`deadline_dispatch_requests` 関数は, 処理対象の並べ替えキューを選択した後, サブキューからディスパッチキューに I/O 要求を移動する `deadline_move_request` 関数を呼び出す．既存の実装では, Read と Write のキューの両方に I/O 要求が存在する場合は，Read のキューを選択する．本機構では，この条件判定に低優先度キューを選択する処理を追加した．図 7 に示す通り，既存の 2 つの並べ替えキューに要求が存在しない場合にのみ，低優先度処理の並べ替えキューが選択されるように実装した.

Deadline I/O スケジューラでは，I/O 要求が追加された時のみディスパッチが行われているが,（2）の仕組みと（3）の仕組みでは，I/O 要求の追加以外のタイミングでもディスパッチを行う必要がある．そのため，カーネルタイマを用いて必要なタイミングでディスパッチを行うようにした．カーネルタイマとは，任意の時刻に，任意の関数を呼び出すことができるカーネルの機能である．また，必要な変数はスケジューラ固有のデータ構造に追加した.

（2）と（3）の仕組みに関する擬似コードを図 8 に示す.

低優先度の I/O 要求が発行されると，その要求を低優先度用の並べ替えキューと期限付きキューに追加した後, ディスパッチのために `deadline_dispatch_requests` 関数が呼び出される．すると,（1）の仕組みだけでは, 通常の I/O 要求が存在しない時には常に低優先度の I/O 要求がディスパッチされる．そこで，`deadline_dispatch_requests` 関数を図 8 のように変更して，バッチモードであるか, 直前に低優先度の I/O 要求をディスパッチしてから `idle_dispatch_interval` の時間経過している場合のみ低優先度 I/O をディスパッチするようにした．同時に，最後にディスパッチした時刻 `idle_dispatched_time` を更

```
deadline_dispatch_requests(struct
    deadline_data *dd) {
  ...
  else if (idles) {
    elapsed_time =
      jiffies - dd->idle_dispatched_time;
    if (elapsed_time >
        dd->idle_dispatch_interval ||
        dd->batch_mode) {
      dd->idle_dispatched_time = jiffies;
      if (!dd->batch_mode)
        set_dispatch_timer();
      data_dir = IDLE;
      goto dispatch;
    }
  }
  ...
    dispatch:
    if (通常の I/O 要求をディスパッチ)
      /* 通常モードへ移行 */
      dd->batch_mode = FALSE;
  ...
}
deadline_completed_request(struct
    deadline_data *dd) {
  if (通常の I/O 要求が完了)
    set_redispatch_timer();
}
on_timer(struct deadline_data *dd) {
  if (idle_redispatch_time 経過) {
    clear_dispatch_timer();
    dd->batch_mode = TRUE;  // バッチモード
    if (!list_empty(dd->sort_list[LOWPR])) {
      deadline_dispatch_requests(dd);
      ストラテジルーチン呼び出し;
    }
  } else if (idle_dispatch_time 経過) {
    if (!list_empty(dd->sort_list[LOWPR])) {
      deadline_dispatch_requests(dd);
      ストラテジルーチン呼び出し;
    }
  }
}
```

図 8 （3）の仕組みに関する擬似コード

新する．バッチモードでない時は，次に低優先度 I/O をディスパッチしてもよいのは idle_dispatch_interval の時間経過後である．現在溜まっている I/O 要求や，次にディスパッチ可能になるまでに発行された I/O 要求が，idle_dispatch_interval の時間経過後すぐにディスパッチされるように，set_dispatch_timer 関数によってタイマもセットしておく．

idle_dispatch_interval の時間が経過すると on_timer 関数が呼び出される．ここでタイマが発火した原因を調べ，batch_mode_delay の時間経過したことによる発火であれば deadline_dispatch_requests 関数を呼び出して，次の低優先度の I/O 要求をディスパッチする．その中で，次のタイマがセットされる．ただし，低優先度 I/O 要求が並び替えキューになければ何もしない．

バッチモードに移行するためのタイマは，通常の I/O 要求が完了した時にセットする．I/O 要求が完了すると，deadline_completed_request 関数が呼び出される．その中で set_redispatch_timer 関数を呼び出すことで設定する．これにより，deadline_redispatch_interval の時間経過後にも on_timer 関数が呼び出されるようになる．deadline_completed_request 関数は I/O 要求が完了する度に呼び出され，その度にタイマをセットし直す．

on_timer 関数でタイマが発火した原因を調べ，deadline_redispatch_interval の時間経過したことによる発火であれば，今度はバッチモードに移行したうえで，deadline_dispatch_requests 関数を呼び出して，溜まっている低優先度の I/O 要求をディスパッチする．バッチモードに移行すると，低優先度の I/O はすぐにディスパッチされるようになるので，idle_dispatch_interval 時間待つためのタイマも解除する．バッチモードは通常の I/O 要求を処理した時に deadline_dispatch_requests 関数の最後で解除され，通常モードに移行する．

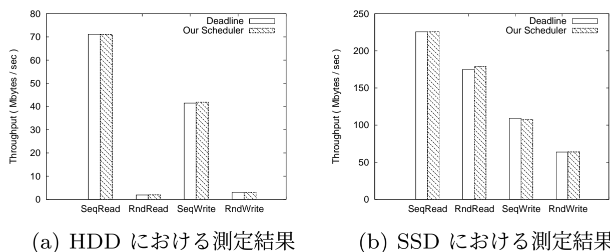
6. 評価

本機構の評価を行うために，既存の Deadline I/O スケジューラとの比較を行うベンチマークテストを行った．ベンチマークテストの実行環境を表 1 に示す．

また，ベンチマークツールは SysBench [7] を用いた．ベンチマークテストは，単純なファイルアクセスの他に，データベースサーバを用いて行った．

表 1 テスト環境

OS	CentOS release 6.5 （Final）
Linux カーネル	3.12.7
CPU	Intel（R）Core i7-3770@3.40GHz，4 コア
メモリ	8 GiB
記憶装置 1	SSD 240 GB （SSDSC2CT240A4）
記憶装置 2	HDD 320 GB，7200 rpm （HDP72503）



(a) HDD における測定結果　(b) SSD における測定結果

図 9 基本性能

6.1 基本性能

I/O スケジューラごとの基本性能の比較を行った．比較のための指標は，1 秒間におけるスループットとした．1 分間に複数のファイルに対して Read/Write を行うベンチマークテストを行い，平均のスループットを測定した．ページキャッシュの影響を受けずに評価を行うために，ベンチマークツールが発行する I/O 処理はすべてダイレクト I/O を用いて行った．アクセス対象のファイルとして 80MB のファイルを 128 個作成した．記憶装置へのアクセスの方法は，以下の I/O 処理の方法を用いた．

Sequential Read/Write
データを先頭から順番に Read/Write をするアクセス方法．

Random Read
必要な部分を直接 Read/Write するアクセス方法．

HDD 上での測定結果を 図 9(a)，SSD 上での測定結果を図 9(b) に示す．それぞれのグラフにおいて，縦軸は 1 秒間毎のスループット量，横軸は アクセス方法を表している．縦軸の値が大きければ大きいほど，大量の Read/Write 処理が可能であることを意味する．それぞれのグラフから，既存の Deadline I/O スケジューラと比較して，本機構はオーバーヘッドがほぼないことが分かった．

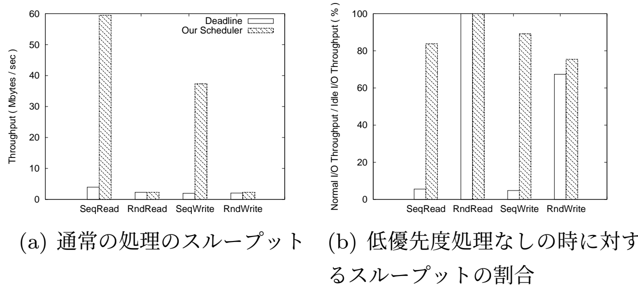
6.2 低優先度処理のスケジューリング

通常の処理と低優先度の処理を同時に実行した場合に，通常の処理がどれだけ処理できるかを比較した．評価は，ファイルアクセスとデータベースアクセスの 2 つのパターンのアクセス方法を用いて行った．

6.2.1 ファイルアクセスによる評価

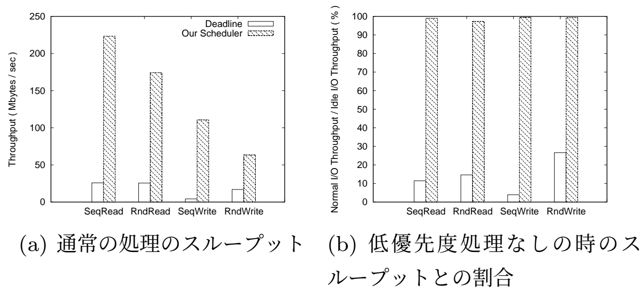
ファイルアクセスによってベンチマークテストを行った．評価の指標は，低優先度処理が行われていない場合のスループットに対する通常の処理のスループットの割合とした．低優先度処理が行われていない場合のスループットは，6.1 節における測定結果を用いた．

ベンチマークテストは，6.1 節の処理と並行して低優先度処理を行った．低優先度処理は，Sequential Read を通常の処理が完了するまで実行し続けるものとした．実行時には，ionice コマンドを用いて低優先度クラスを指定し



(a) 通常の処理のスループット　(b) 低優先度処理なしの時に対するスループットの割合

図 10　HDD における測定結果



(a) 通常の処理のスループット　(b) 低優先度処理なしの時のスループットとの割合

図 11　SSD における測定結果

て実行した．

HDD 上での測定結果を図 10，SSD 上での測定結果を図 11 に示す．それぞれの図の左のグラフにおいて，縦軸は 1 秒間毎のスループット，横軸は アクセス方法を表している．スループットの値が大きければ大きいほど，大量の Read/Write 処理が可能であることを意味する．それぞれの図の右のグラフにおいて，縦軸は低優先度処理が行われていない場合のスループットに対する通常の処理のスループットの割合，横軸はアクセス方法を表している．この割合が大きければ大きいほど，低優先度処理による悪影響がなく，通常の Read/Write 処理が可能であることを意味する．図 10(a) から，HDD に対する Random Read/Write の場合は，2 つのスケジューラの振る舞いに大きな差がないことが分かる．しかし，それ以外のアクセス方法においては，既存の Deadline I/O スケジューラを用いた場合に，スループットが極端に低下している．図 10(b) と図 11(b) の結果から，既存の Deadline I/O スケジューラでは，低優先度処理なしの時と比較して 70%以上スループットが低下している．一方，本機構を用いた場合のスループットの低下は，HDD では 25%以下，SSD では 2.75 % 以下に抑えられている．これらの結果から，通常の処理と低優先度処理を並行して行った場合，本機構では，スループットの低下を抑制することができることが分かる．

6.2.2　データベースアクセスによる評価

データベースアクセスによってベンチマークテストを行った．一般的にデータベースの性能比較の場合，スループットではなく *TPS*（*Transaction Per Second*）という単位を用いて比較を行う．TPS とは，1 秒当たりのトランザクション処理件数を指す単位である．そのため，評価の指

```
query_cache_size = 0
innodb_buffer_pool_size = 0
innodb_flush_method = O_DIRECT
```

図 12　MySQL の設定

```
SELECT <FIELD_NAME> FROM <TABLE_NAME> WHERE id
    =<ID>;
```

図 13　1 トランザクション中に実行される SQL

標は低優先度処理が行われていない場合の TPS に対する低優先度処理が行われている場合の TPS の割合とした．

データベースサーバは，シェアの高い MySQL を用いた．バージョンは 5.5，ストレージエンジンは InnoDB を用いた．今回のベンチマークテストでは，1 つのテーブルに対してアクセスを行った．テーブル内のレコード数は 4400 万であり，データベース全体のサイズは 10 GB とした．

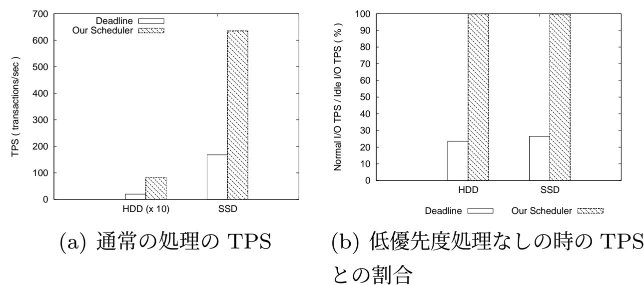
ディスクアクセスをバイパスせずにベンチマークテストを行うために，MySQL のキャッシュ機構を無効にした．また，ページキャッシュや遅延書き込みが発生しないようにするために，`O_DIRECT` で I/O 処理を行うように設定した．図 12 に MySQL の設定を示す．MySQL に対して以下の設定を行った．これらの設定によって，メモリより大きいサイズのデータを保持するデータベースサーバへのアクセスに近い状況を実現した．これは，データサイズの急激な肥大化により，シャーディングやスペックアップなどの対策が間に合わない場合などに実際に起こりうる状況である．

また，CPU がボトルネックにならないように，1 トランザクション処理中に実行される SQL は単純なものにした．1 トランザクション処理中に実行される SQL 文を図 13 に示す．ベンチマークテストでは，1 トランザクション処理中に，図 13 の SQL 文の <FIELD_NAME>，<TABLE_NAME>，および <ID> を適切な値に置き換えたクエリ が 10 回行われる．

データベース処理と並行して低優先度処理を行った．6.1 節と同様に，低優先度処理は Sequential Read を通常の処理を完了するまで実行し続けるものとした．

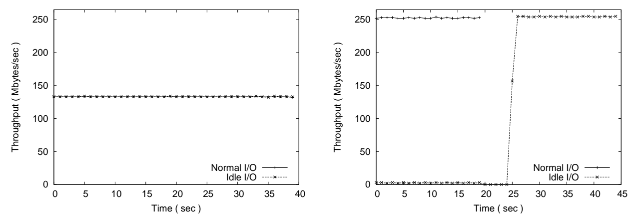
低優先度処理をデータベース処理と並行して実行していない場合と，並行して実行した場合を比較した結果を図 14 に示す．図 14(a) において，縦軸は 1 秒間毎の TPS を表している．TPS の値が大きければ大きいほど，多くのトランザクション処理が可能であることを意味する．なお，HDD は SSD に比べて遅かったため，図 14(a) には 10 倍した値を載せた．図 14(b) において，縦軸は低優先度処理が行われていない場合の TPS に対する通常の処理の TPS の割合を表している．この割合が大きければ大きいほど，低優先度処理による悪影響がなく，通常の Read/Write 処理が可能であることを意味する．また，それぞれのグラフ

© 2014 Information Processing Society of Japan



(a) 通常の処理の TPS　(b) 低優先度処理なしの時の TPS との割合

図 14　SSD における測定結果



(a) Deadline I/O スケジューラにおける測定結果　(b) 本機構における測定結果

図 15　低優先度処理のスループットの様子

において，横軸は記憶装置を表している．図 14(b) から，既存の Deadline I/O スケジューラでは，低優先度処理なしの時と比較して TPS が 70 %以上低下していることが分かる．一方，本機構を用いた場合は，HDD と SSD の両方の記憶装置において，TPS の低下を 1% 以下に抑えている．これらの結果から，通常の処理と低優先度処理を並行して行った場合，本機構では，TPS の低下を抑制することができることが分かった．

6.3　低優先度処理のスループット

設計通りに低優先度処理がディスパッチされているかを確認するために，ファイルアクセスを用いたベンチマークテストを行い評価をした．それぞれの I/O スケジューラに対して，5GB のファイルに対して Sequential Read を行う処理を 2 並列で実行した．一方の処理は，優先度指定なしで行い，もう一方の処理は，低優先度指定をして実行した．本機構を用いる場合には，`idle_dispatch_interval` と `batch_mode_delay` の値を，それぞれ 200 （ms） と 5000 （ms） に設定した．記憶装置は SSD を用いた．

ベンチマークテストの結果を図 15 に示す．図 15 のそれぞれのグラフにおいて，縦軸はスループット，横軸はベンチマークテスト開始時刻からの経過時間を表している．図 15(a) から，既存の Deadline I/O スケジューラでは，スループットを 2 つのプロセスが半分ずつ分けあって処理をしていることが分かる．どちらの処理も完了までに 40 秒程度経過している．図 15(b) から，本機構では，通常の処理が SSD のスループットのほとんどを使い，低優先度処理は，通常の処理が完了するまでは 2〜3MB/sec 程度のスループットを使っている．通常の処理は 20 秒程度で完了し，低優先度処理は 45 秒程度で完了している．また，通常の処理が完了した後，低優先度処理のスループットが 0 である時間が 5 秒続いている．これは，`batch_mode_delay` の値に一致している．通常の処理が完了して 5 秒間経った後，スループットを使い切って処理を行っている．これにより，バッチモードが正常に動作していることが分かった．

また，`idle_dispatch_interval` の値を変化させながら，上記と同様のベンチマークテストを行った．低優先度処理の平均スループットを図 16 に示す．なお，平均スループットの計算対象は，低優先度処理は通常の処理と並行して実行している期間，すなわち，通常モード時のスループットのみとした．図 16 のグラフにおいて，縦軸はスループット，横軸は `idle_dispatch_interval` の値を表している．図 16 から，`idle_dispatch_interval` の値が大きくなればなるほど，スループットが低下していることが分かる．これは，低優先度処理のディスパッチを行う間隔が大きくなったためであると考えられる．このことから，ユーザは `idle_dispatch_interval` の値を，低優先度処理のスループットを大きくしたい場合は小さくし，スループットを小さくしたい場合は大きくすればよいということが分かる．

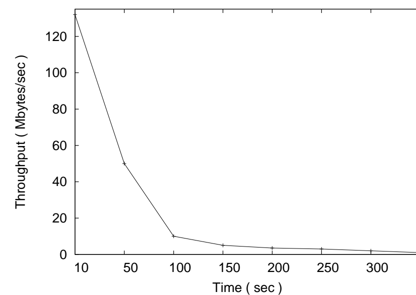


図 16　idle_dispatch_interval と スループットの関係

7.　おわりに

本論文は，低優先度を指定可能なリアルタイム処理向けの I/O スケジューラを提案した．本機構は，通常の処理と低優先度処理が共存した場合に，低優先度処理のスループットを抑えることで，通常の処理を優先的に実行することを可能にする．リアルタイム処理向けの I/O スケジューラである Deadline I/O スケジューラに対して，低優先度処理のスループットを抑える機能を追加することで実装を行った．ベンチマークテストを用いた評価の結果から，既存の Deadline I/O スケジューラではデータベースと関係のない I/O 処理の実行中に TPS が 70% 以上低下したのに対し，提案する I/O スケジューラでは 25% 以下の低下に抑えられることを確認した．

参考文献

[1] Dunn, M.: A New I/O Scheduler for Solid State Deveces (2010).

© 2014 Information Processing Society of Japan

[2] Dong, R.: TPPS: Tiny Parallel Proportion Scheduler (2013), https://lkml.org/lkml/2013/6/4/949.
[3] Seelam, S. and Romero, R.: Enhancements to Linux I/O Scheduling (2005).
[4] Nou, R. and Giralt, J.: Automatic I/O scheduler selection through online workload analysis (2010), IEEE.
[5] Lunde, C. H.: Improving Disk I/O Performance on Linux, Master's thesis, Hvard Espeland (2009), http://agesage.co.jp/.
[6] Menage, P.: CGROUPS: https://www.kernel.org/doc/Documentation/cgroups/cgroups.txt.
[7] Kopytov, A.: SysBench manual (2009), http://sysbench.sourceforge.net/docs/.

© 2014 Information Processing Society of Japan