平成 25 年度

卒 業 論 文

情報・通信工学科
コンピュータサイエンスコース

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

指導教員 : 鵜川始陽 助教
学籍番号 : 1011121
氏名 : 髙村成道

提出月日 : 平成 26 年 1 月 31 日 (金)

要旨

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．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 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た．

目次

1 **はじめに** 1
- 1.1 本研究の背景 . . . 1
- 1.2 本研究の目的 . . . 2
- 1.3 本論文の構成 . . . 2

2 **ファイル I/O の仕組み** 3
- 2.1 ファイル I/O の概観 . . . 3
- 2.2 Read と Write . . . 5
- 2.3 I/O スケジューラ . . . 9

3 **設計** 14
- 3.1 方針 . . . 14
- 3.2 既存の Deadline I/O スケジューラ . . . 16
- 3.3 低優先度キュー付き Deadline I/O スケジューラ . . . 17
- 3.4 低優先度の I/O 要求のディスパッチ . . . 18

4 **実装** 21
- 4.1 既存の Deadline I/O スケジューラの実装 . . . 21
- 4.2 低優先度キューの定義 . . . 21
- 4.3 ディスパッチの制御 . . . 22
- 4.4 期限の設定 . . . 28
- 4.5 I/O 優先度の取得 . . . 28
- 4.6 設定値のパラメタ化 . . . 29

5 **評価** 30
- 5.1 基本性能の比較 . . . 30
- 5.2 低優先度処理のスケジューリング . . . 31
- 5.3 低優先度処理のスループット . . . 35

6 **関連研究** 37

7 **おわりに** 38

1 はじめに

1.1 本研究の背景

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．多くの場合，Web サービスのバックエンドではデータベースサーバが稼働している．データベースサーバとは，複数のサーバが役割分担をしているようなシステムにおいて，データベース管理システムが稼動しているデータベースを内部に持つサーバのことである．代表的なデータベースサーバには，商用のものでは Oracle (日本オラクル) や SQL Server (マイクロソフト)，オープンソースものでは PostgreSQL や MySQL などがある．データベースサーバの役割は，クライアントのリクエストに応じて，データベース上のデータの参照，書き換えや削除などの処理を行い，処理結果をアプリケーションに返すことである．クライアントでは，データベースサーバに対してリクエストを送信してからレスポンスが返ってくるまでが待機時間となる．つまり，データベースサーバ上の処理が遅延すればするほど，クライアントの待機時間も長くなる．一般的にデータベースサーバでは，数 GB 〜 数 TB 規模のデータを扱い，これらの大量のデータを高速に処理することが求められている．データベースサーバには，UNIX 系オペレーティングシステムが用いられることが多く，その中でも Linux 系 OS のシェアは高い．

データベースサーバのメンテナンスは， Web 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として，データの集計や削除等が挙げられる．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．一般的に，負荷は以下の 2 種類に分類される．

- CPU 負荷
- I/O 負荷

CPU 負荷はディスクの入出力 (Input/Output，以下 I/O) は行わないが大量の計算処理が必要な場合に生じる．一方，I/O 負荷はディスクに対して大量に読み書きを行う場合に生じる．データベースサーバでは，CPU での計算より I/O 処理に伴うディスクアクセスの方が時間がかかり，大量のデータを扱う大規模サーバになればなるほど I/O に依存する割合が大きくなる．そのため，I/O 負荷によるサービスへの影響は深刻化する．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と，メンテナンスのために大量の I/O 処理が行われデータベースの処理が長時間停止するという深刻な状況が生じる．

メンテナンス処理の影響を最小限に抑えるための策として，サーバの高性能化が挙げられる．しかし，この方法の場合サーバの負荷が高くなる度に高性能化が必要となってしまい，根本的な解決にはならいない．そもそもメンテナンス処理のためだけに高性能化をするのは，現実的ではない．もう一つの対策として，メンテナンスの I/O 処理の実行をゆっくり行

うことが挙げられる．Linux において，CPU 処理の優先度指定は nice コマンドを使って可能であるが，I/O 処理の優先度は I/O スケジューラに依存する．データベースサーバでは，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．

1.2 本研究の目的

本研究では，Linux の データベースサーバを対象とし，オンラインメンテナンスによるサービスの停止を防ぐことを目的とする．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，実装する．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をすることで評価を行う．

1.3 本論文の構成

本論文では，2 章でファイル I/O について説明する．3 章と 4 章で本研究で実装する I/O スケジューラについての設計と実装について述べ，5 章で実装した I/O スケジューラの評価を行う．6 章で関連研究について述べ，最後に，7 章で本論文をまとめる．

2 ファイル I/O の仕組み

Linux におけるファイル I/O は，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．本章では，それらのカーネル要素の説明並びに，ファイル I/O の仕組みを説明する．

2.1 ファイル I/O の概観

ファイル I/O とは，ファイルのデータをユーザ空間に読み込んだり，ユーザ空間のデータをファイルに書き込んだりする操作のことである．この操作は，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．この概略を図 1 に示す．ファイル I/O に関するカーネル空間のサービスルーチンは，次の要素から構成されている．

VFS (Virtual File System)
: カーネル内のソフトウェアレイヤで，ユーザ空間のプロセスにファイルシステムのインターフェースを提供するための仕組みである．これにより，異なるファイルシステムを統一的なインターフェースで扱うことができる．VFS 機能の一部を以下に示す．

  - `open`，`read`，`write`，`close` などのシステムコールを通じて，各ファイルへのアクセスを提供する．
  - デバイスに格納されたデータをデバイスやファイルシステムの種類によらず，木構造上に存在するディレクトリ内のファイルとして見せる．

  ハードウェアデバイスがセクタをデータ転送の基本単位として扱うのに対し，VFS ではブロックを基本単位とする．ブロックは 1 つ以上の隣接したセクタに対応する．以下の 2 つは，VFS が提供する機能の一部である．

  ファイルシステム
  : 様々な物理デバイス上のデータを「ファイル」として OS から透過的に扱うための機能である．Microsoft 社の NTFS (Windows NT 4 以降) や Linux の EXT4 などが例として挙げられる．

  ディスクキャッシュ
  : 通常はディスクに保存されるデータをシステムがメモリ上に保持できるようにするソフトウェア機構である．この機構により，あとで同じデータにアクセスする場合，ディスクキャッシュにデータが存在すれば，ディスクアクセスすることなく処理が行われるため，高速なアクセスが可能となる．

汎用ブロック層
: システムに接続されたブロック型デバイスに対して出されたすべての要求を処理する

カーネル要素である．どのブロック型デバイスでも共通して利用できるデータ構造を用いることで，各ハードウェアの特性を隠蔽し，ブロック型デバイスを抽象的に扱うことができる．

I/O スケジューラ層

ディスクのヘッドの平均移動回数を減らすために，汎用ブロック層から届いた I/O 要求を並び替え，物理的な媒体上で近くに位置するデータへの要求をまとめる．まとめられた要求はディスパッチキューというキューに入れられ，デバイスドライバに送られる．この処理により，効率的に I/O 処理が行われる．異なるスケジューリングアルゴリズムを実装した複数の I/O スケジューラがあり，目的に応じて使い分けることができる．

デバイスドライバ

ディスパッチキューに存在する要求に従って，物理デバイスにデータ転送を行う．

物理デバイス

Linux カーネルでは，物理デバイスをテープデバイスなどのシーケンシャルアクセスのみを提供するキャラクタ型デバイスと，ハードディスクなどのランダムアクセスが可能なブロック型デバイスの 2 種類に区別して扱う．本研究では，ブロック型デバイスのみを対象とする．通常，物理デバイスには 512 Byte のセクタ単位でデータが書き込まれる．

図 1 のディスクキャッシュと物理デバイス間のデータ転送は，ユーザ空間のデータを返す時に同期的に行われる場合と，カーネルスレッドによって非同期に行われる場合の 2 つのパターンがある．

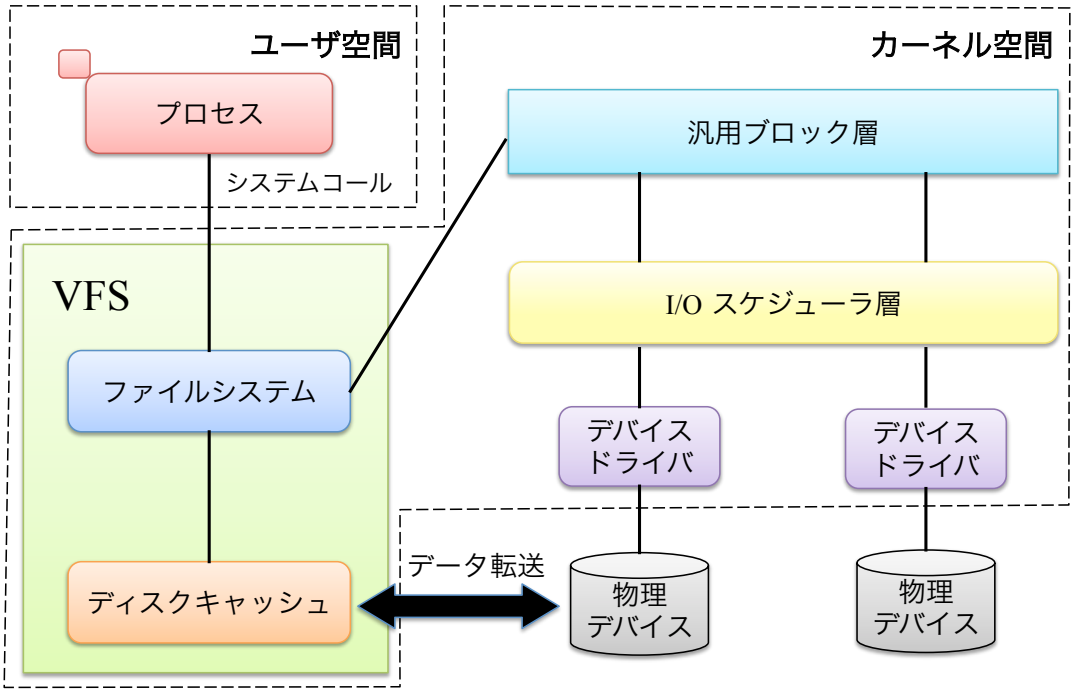


図 1　Linux におけるファイル I/O の概略

2.2　Read と Write

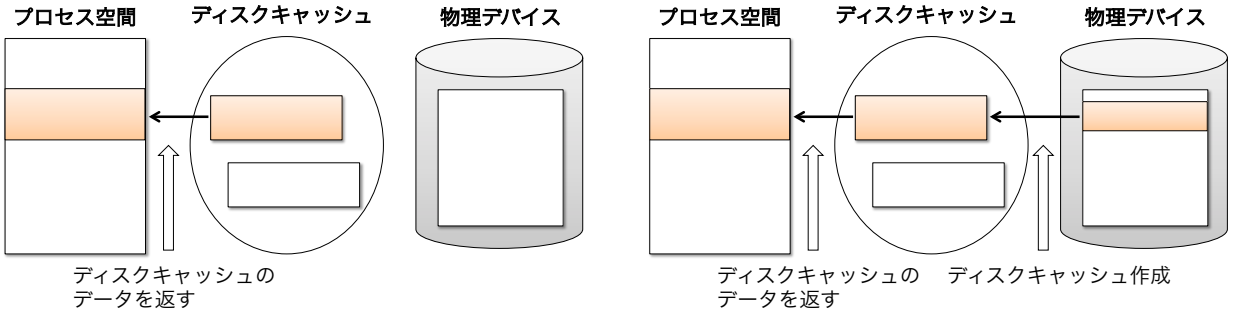
2.2.1　Read

`read` システムコールを発行した場合，カーネルは次に示す手順でこの要求を処理する．処理の様子を図 2 に示す．

1. `read` システムコールにより，適切な VFS の関数を呼び出しファイルディスクリプタとファイルのオフセットを渡す．
2. VFS の関数は，要求を受けたデータがディスクキャッシュに存在するかを確認し，要求したデータがディスクキャッシュに存在する場合は，ブロック型デバイスへはアクセスせずに，ディスクキャッシュ上のデータをユーザプロセスへ返して終了する．ディスクキャッシュにデータが存在しない場合は，ファイルシステム固有の関数を呼び出すことで，データの物理的な位置を特定し，ブロック型デバイスに対して読み込み操作を発行する．
3. 読み込み操作の発行には汎用ブロック層を使用する．一般的に 1 回の I/O 操作では，ディスク上で隣接する「領域」または「データ」を扱う．ただし，要求を受けたデータは必ずしもディスク上で隣接しているとは限らないため，その場合には，汎用ブロック層は I/O 処理を何回か実行する．
4. 汎用ブロック層の下では，I/O スケジューラがスケジューリングアルゴリズムに従って，保留中の I/O データの転送要求を並べ替え，併合した後にディスパッチキューに

移動する．

5. デバイスドライバは，ディスパッチキューに存在する要求をもとに，ディスクコントローラのハードウェアインターフェースへ適切なコマンドを送ることでデータ転送を行う．ディスクから読み取ったデータは，汎用ブロック層を経由して VFS に転送される．
6. VFS 関数にてディスクキャッシュを作成する．その後，ディスクキャッシュ上のデータをユーザに返す．



(a) ディスクキャッシュ上にデータが存在する場合

(b) ディスクキャッシュ上にデータが存在しない場合

図 2　Read の動作

2.2.2　Write

`write` システムコールを発行した場合，カーネルは次に示す手順でプロセスの要求を処理する．処理の様子を図 3 に示す．

1. `write` システムコールにより，適切な VFS 関数を呼び出しファイルディスクリプタとファイルのオフセットを渡す．
2. VFS の関数は，要求を受けたデータをディスクキャッシュに書き込む．
3. ディスクキャッシュの使用状況から物理デバイスへのデータ転送 (書き込み) が必要か判断する．必要に応じて，ディスクキャッシュのデータを物理デバイスに書き込むカーネルスレッドを起動する．ユーザプロセス側の処理はここで終了する．
4. カーネルスレッドが汎用ブロック層を経由して，I/O スケジューラのリクエストキューに書き込み要求を追加する．
5. 汎用ブロック層の下では，I/O スケジューラがスケジューリングアルゴリズムに従って，保留中の I/O データの転送要求を並べ替え，併合した後にディスパッチキューに移動する．
6. デバイスドライバは，ディスパッチキューに存在する要求をもとに，ディスクコントローラのハードウェアインターフェースへ適切なコマンドを送ることでデータ転送を行う．

上述の書き込み処理は，ユーザ空間のデータをファイルキャッシュにコピーした時点でユーザ空間に復帰し，ディスクへの書き出しはカーネルスレッドによって後で行われるため，**遅延書き込み**と呼ぶ．ディスクへの書き出しを待たずにユーザ空間に復帰するため，見かけ上の性能は良くなる．ただし，`write` システムコールが復帰しても，その時点で物理デバイスへの書き込みが完了しているという保証はない．そのため，システムが停止すると書き込んだはずのデータが物理デバイスに書き込まれずに消えてしまう場合がある．これを防ぐために，カーネルスレッドが定期的に書き込みを行っている．また，ディスクキャッシュにデータを書き込んだ直後に，ディスクキャッシュのデータをディスクに書き出す処理を**同期書き込み**と呼ぶ．これは，`O_SYNC` フラグを立ててファイルが開かれている，またはファイルシステムを `sync` オプション付きでマウントされている場合に行われる．

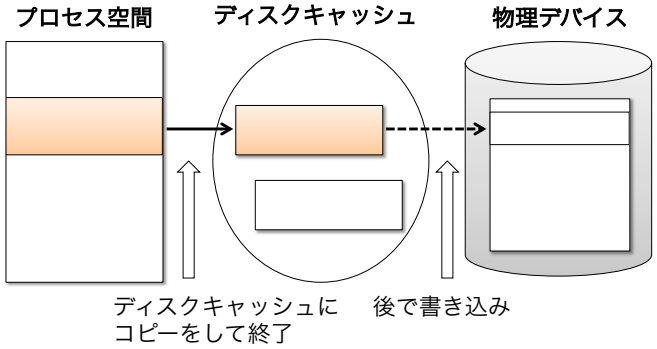


図 3　Write の動作

2.2.3　ダイレクト I/O

ディスクキャッシュ経由の Read/Write は，同じキャッシュに何回もアクセスするようなケースでは有効である．しかし，一度きりしかアクセスしない場合や独自にキャッシュを保持しているアプリケーションを利用する場合は，ディスクキャッシュは役に立たない．アプリケーションの例としては，データベースソフトウェアが挙げられる．これらの問題の対策として，Linux では ダイレクト I/O という機構を提供している．ダイレクト I/O とは，ディスクキャッシュを使わずに，ユーザ空間と物理デバイスの間で直接読み書きをする処理のことである．この機構を用いることで，ディスクキャッシュ経由の Read/Write をする時と比べて，メモリのコピーを 1 回減らすことができる．ダイレクト I/O を用いた Read/Write の様子を図 4 に示す．

Read 時にダイレクト I/O を行う場合，カーネルは次に示すようにしてプロセスの要求を処理する．

1. `O_DIRECT` というフラグを指定して，ファイルをオープンする．これにより，Read がダイレクト I/O を利用して行われるようになる．
2. VFS の関数は，要求を受けたデータが利用可能状態であるかを確認し，ファイルシステム固有の関数を呼び出すことで，データの物理的な位置を特定し，ブロック型デバ

イスに対して読み込み操作を発行する．このとき，O_DIRECT フラグが立っているかを調べる．フラグが立っていれば，ダイレクト I/O を行うための関数を用いて，汎用ブロック層に対して読み込み操作を発行する．

3. 汎用ブロック層の下では，I/O スケジューラがスケジューリングアルゴリズムに従って，保留中の I/O データの転送要求を並べ替え，併合した後にディスパッチキューに移動する．
4. デバイスドライバは，ディスパッチキューに存在する要求をもとに，ディスクコントローラのハードウェアインターフェースへ適切なコマンドを送ることでデータ転送を行う．ディスクから読み取ったデータは，汎用ブロック層を経由して VFS に転送される．
5. VFS の関数を用いて，ページキャッシュを介さずにデータをユーザに返す．

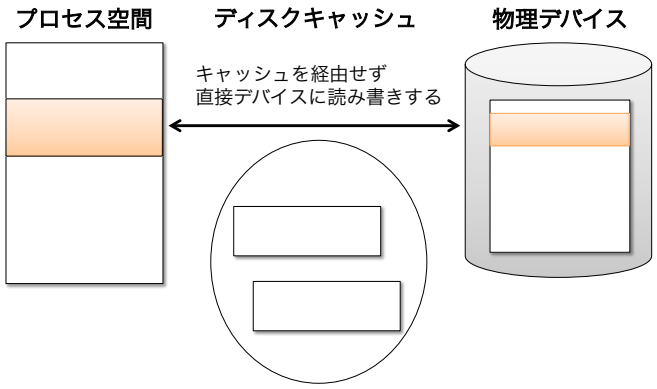


図 4　ダイレクト I/O を用いた Read/Write の動作

2.3 I/O スケジューラ

2.3.1 動作概念

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い，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 5 に示す．I/O スケジューラ層は，以下の 2 種類の構造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っている．

`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 スケジューラ層とその前後の層の一部は，`request_queue` という構造体で実現されている．`request_queue` 構造体はブロック型デバイスごとに割り当てられる．`request_queue` 構造体のメンバの一部の説明を以下に示す．

`queue_head`
: ディスパッチキューの先頭要素を示す．

`elevator_queue`
: I/O スケジューラへのポインタである．

`make_request_fn`
: `bio` 構造体を受け取る関数へのポインタである．

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 要求を処理するキューである．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 要求の追加されると呼び出される．

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 構造体同士は併合される．
2. make_request_fn 関数を呼び出し，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．物理的な位置が近い場合は，新たに request 構造体は作成せずに既存の request 構造体 に bio 構造体を併合する．
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，request_fn 関数を呼び出すことで，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．

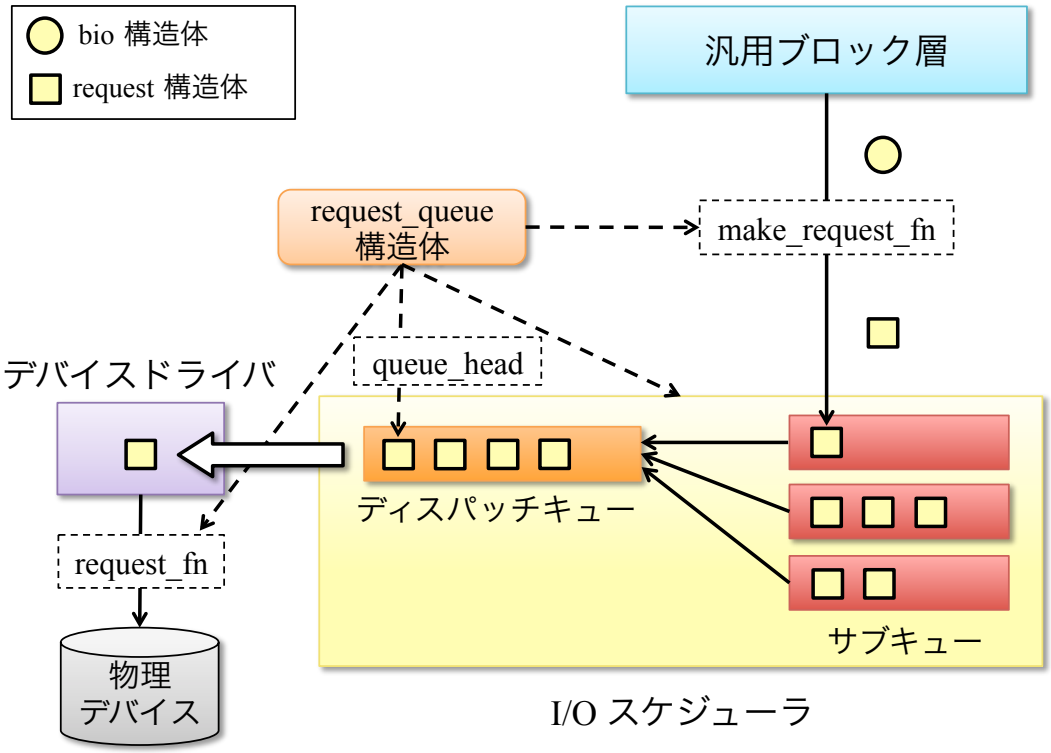


図 5　I/O スケジューラの動作概要

2.3.2 既存の I/O スケジューラ

以下に既存の I/O スケジューラを示す．

NOOP (No Operation)
: 最も単純な I/O スケジューラである．受け取った要求の物理的な位置に応じてディスパッチキューの先頭，または末尾に追加する．

CFQ (Completely Fair Queuing)
: I/O 要求を発行したプロセスごとに I/O 処理時間を公平に割り当てるスケジューラであり，優先度指定が可能である．多くの Linux ディストリビューションにおいて，この I/O スケジューラがデフォルトとして用いられている．

Deadline
: 近隣の I/O 要求を優先して処理する I/O スケジューラである．すべての I/O 要求に対して，期限を付与することで，各 I/O 要求のレイテンシの上限を保証している．そのため，他のスケジューラと比べてリアルタイム処理に向いている．RDBMS (リレーショナルデータベースマネージメントシステム) である Oracle Database や分散ストレージシステムである DRBD (Distributed Replicated Block Device) を用いる場合に推奨されている [1] [2].

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．特定プロセスに対して高優先度の指定も可能ではあるが，現状の実装では最高優先度以外のプロセスの I/O 処理が行われなくなることが多いため実用的ではない．

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．

2.3.3 I/O スケジューラ API

I/O スケジューラでは，API として関数ポインタのテーブルが用意されている．このテーブルは，I/O スケジューラの各メソッドを抽象化しており，それぞれ適切な関数を登録することで，I/O スケジューラにアルゴリズムを実装することができる．たとえば，`request` をディスパッチキューに追加する際に呼び出される関数を定義した場合，その関数のアドレスを `elvator_add_req_fn` ポインタに設定する．以下に，I/O スケジューラ API に定義されている主な関数とその役割を示す．

`elevator_init_fn`

I/O スケジューラが登録された時に呼び出される．I/O スケジューラインスタンスの初期化を行う．I/O スケジューラ内で保持する様々なデータはこの関数によって初期化される．

elevator_add_req_fn
: 新しい I/O 要求をサブキュー に追加する．

elevator_dispatch_fn
: ディスパッチを行う場合に呼び出される．サブキューに存在する I/O 要求をディスパッチキューに追加するための関数．一度の呼び出しで 1 つの I/O 要求 をディスパッチする．

elvator_completed_req_fn
: デバイスドライバが 1 つの I/O 要求を完了する度に呼び出される．

3 設計

本章では，本機構の設計について述べる．既存の Deadline I/O スケジューラに対する拡張の方針とその詳細を説明する．

3.1 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．

処理性能の維持

リアルタイム処理向けの I/O スケジューラであることを維持するため，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．

低優先度 I/O 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．なお，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，ディスクの性能を最大限に引き出すために，低優先度の I/O 要求であってもスループットを制限せずに処理する．

期限の設定

低優先度 の I/O 処理についても，他の I/O 処理と同様に期限を設定する．ここでは，レイテンシの保証ではなく，単に安全装置としての役割を意図している．そのため，低優先度 の I/O 要求の期限は，他の I/O 要求に設定されている期限より長めに設定する．

設定値のパラメタ化

期限の設定やスループットの制限などは，ワークロードやマシンの環境によって最適値が異なることが想定される．そのため，それらを決定する変数をパラメタ化をして，ユーザがシステムが起動している場合でも動的に変更できるようにする．

LKM (Loadable Kernel Module) による実装

LKM とは，オペレーティングシステムの動作中のカーネルを拡張することのできる拡張モジュールのことである．通常，Linux カーネルを拡張した場合，その機能を適用するにはカーネルの再コンパイルとシステムの再起動が必要となる．これは導入の際に大きな障壁となる．そこで，本機構の導入コストを下げるために LKM を用いて実装を行う．これにより，ユーザは動作中のシステムに対して本機構を導入できるよ

うになる.

3.2 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 6 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，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することで，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．もし一定時間処理されていない要求があれば，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 6 に示す．図 6 において，左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 6 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される．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．

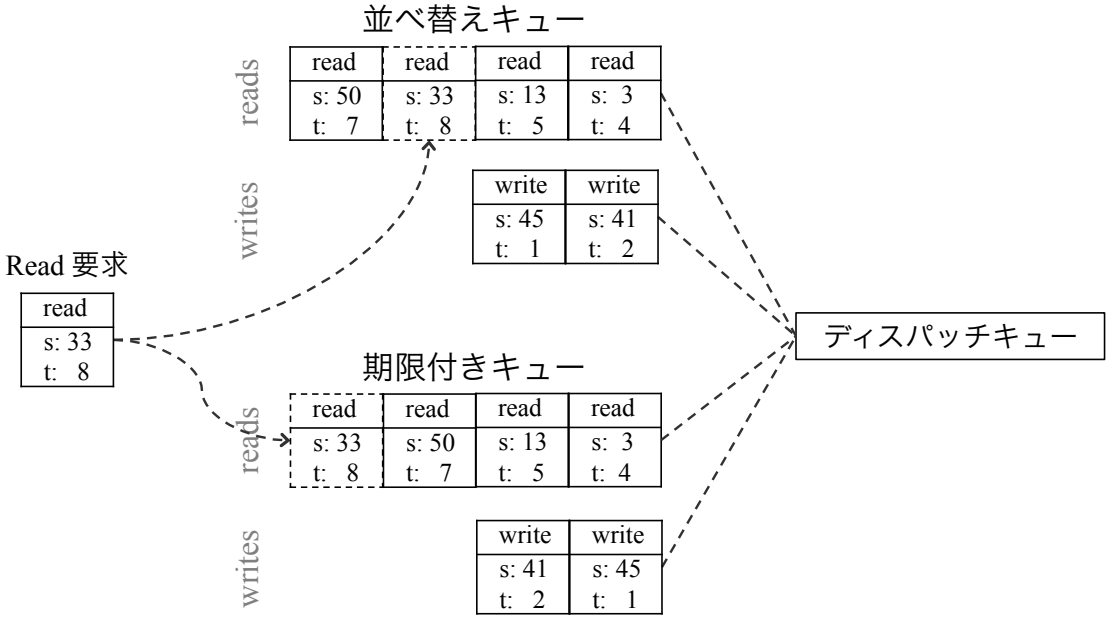


図 6　Deadline I/O スケジューラにおける既存キュー

3.3 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，それぞれ 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 7 に示す．図 7 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 7 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．低優先度キューでは，Read と Write は区別しない．低優先度の I/O 要求に関するディスパッチの動作については次節で説明する．

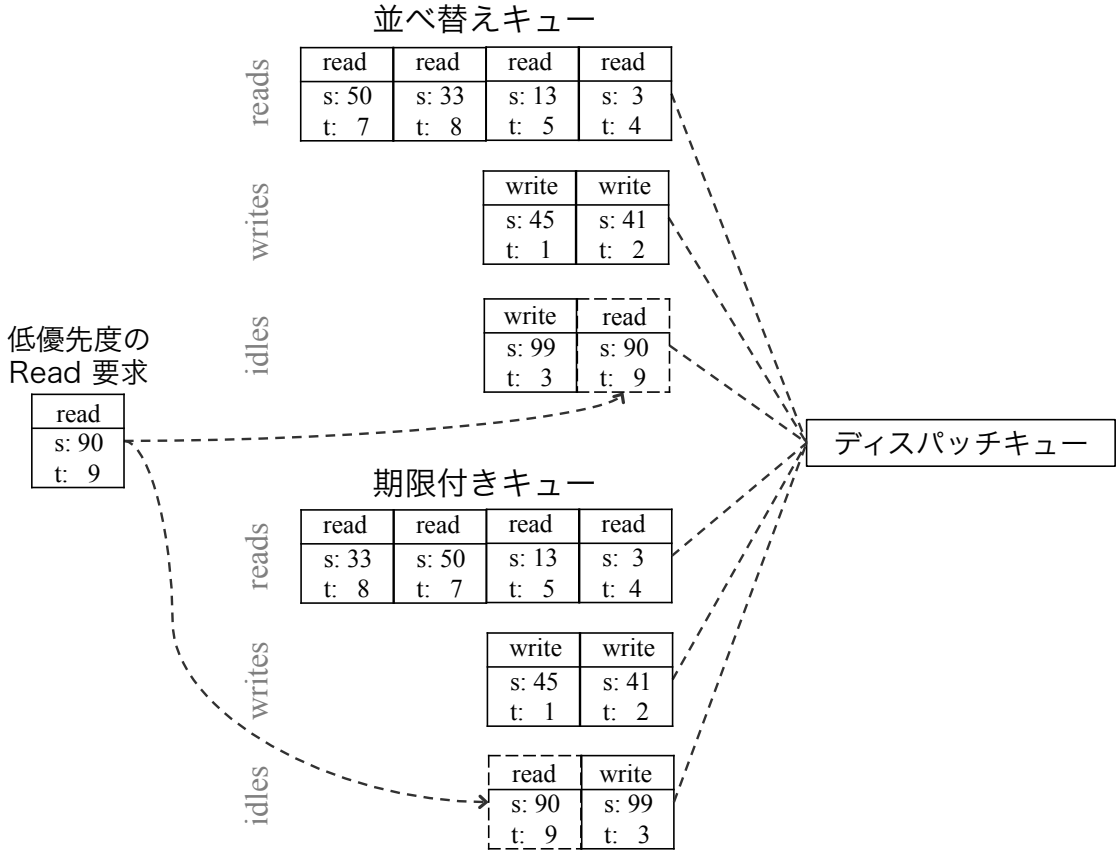


図 7　拡張後の Deadline I/O スケジューラ

3.4 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を工夫し低優先度の I/O 処理のスループットを制限することで，低優先度処理を実現する．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 要求がサブキューに存在しない場合に，ディスパッチを行う．
2. 低優先度の I/O 要求は，スループットを制限するために一定時間間隔でディスパッチを行う．
3. 低優先度 の I/O 要求のみ存在する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処理を行う．

1 つ目は，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チされる．たとえば，図 7 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 8 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 つ目は，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．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．つまり，一度のディスパッチで必ずサブキューは空となる．1 つ目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，すぐさま低優先度の 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，通常の I/O 処理と低優先度の I/O 処理のスループットが同程度になってしまう．そこで，本機構では意図的にディスパッチのタイミング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，スループットを大幅に抑えることができるようになる．この一定時間を `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 8 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 8 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．実装では，`idle_dispatch_interval` をパラメタ化し，ユーザ定義が可能な値とする．

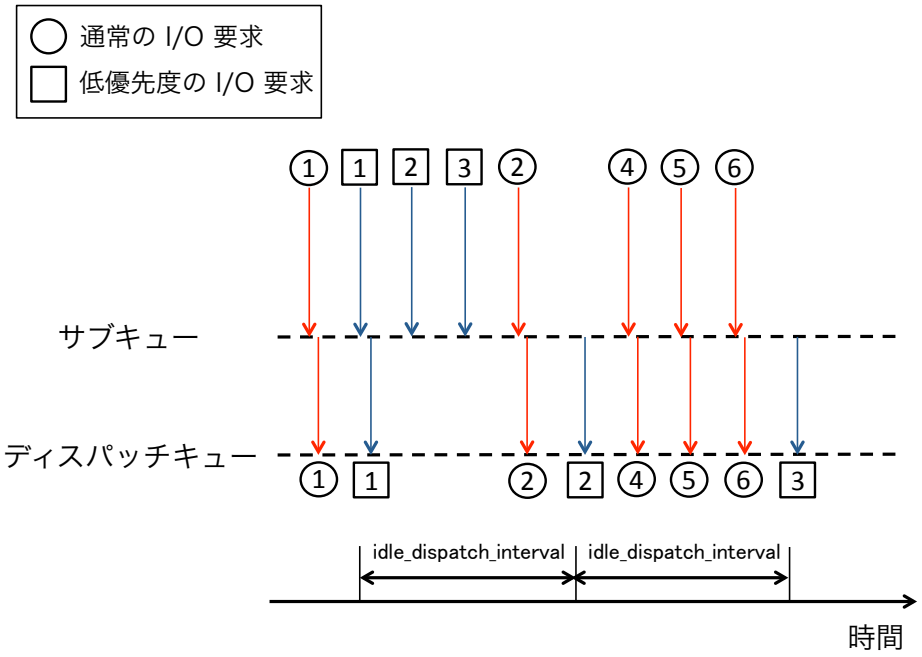


図 8　低優先度の I/O 要求のディスパッチ間隔

3 つ目は，低優先度の I/O 処理の際にマシン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 つ目の仕組みで導入したディスパッチ制御は解除される．これにより，低優先度の I/O 処理が通常の I/O 処理と同程度のスループットで処理が行われることになる．バッチ処理の最中に通常の I/O 要求が追加された場合は，1 つ目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た後，通常モードに移行する．バッチモードへの移行は，I/O 処理が完了してから一定時間経過したあと，サブキューに低優先度の I/O 要求が存在している場合に行う．この一定時間を `idle_redispatch_interval` と定義する．それぞれのモード移行の動作を図 9 に示す．`idle_redispatch_interval` 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．実装では，`idle_redispatch_interval` をパラメタ化し，ユーザ定義可能な値とする．

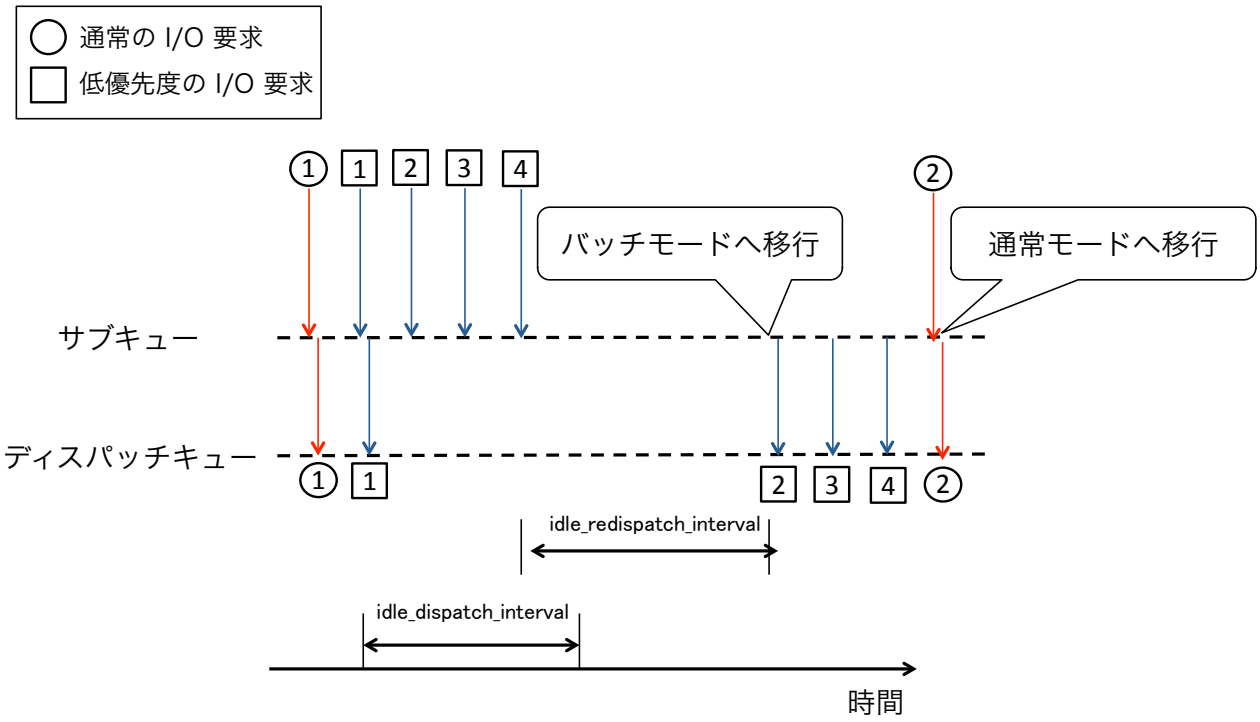

通常の I/O 要求
低優先度の I/O 要求
バッチモードへ移行
通常モードへ移行
サブキュー
ディスパッチキュー
idle_redispatch_interval
idle_dispatch_interval
時間

図 9　モード移行の動作

4 実装

本章では，本機構の実装について述べる．Linux カーネル 3.12.7 に対して，3 章の設計に基づき本機構を実装した．

4.1 既存の Deadline I/O スケジューラの実装

I/O スケジューラ API と Deadline I/O スケジューラの主な関数との対応を表 1 に示す．

I/O スケジューラ API の関数	Deadline I/O スケジューラの関数
`elevator_init_fn`	`deadline_init_queue`
`elevator_add_req_fn`	`deadline_add_request`
`elevator_dispatch_fn`	`deadline_dispatch_requests`
`elevator_completed_fn`	未定義

表 1　I/O スケジューラ API と DeadlineI/O スケジューラの関数との対応

各 I/O スケジューラは，固有のデータを保持している．Deadline I/O スケジューラ固有のデータは，`deadline_data` 構造体で定義されている．この構造体では，各キューの定義や期限を表す変数など，I/O スケジューラ全体に関わるデータがメンバ変数として定義されている．I/O スケジューラ API の関数の多くは，引数としてこの固有のデータ構造を受け取る．設計で挙げた変数の多くは，この構造体のメンバ変数として定義することで実装を行う．

4.2 低優先度キューの定義

設計に基づき低優先度 I/O 要求を追加するためのキューを定義した．サブキューの定義の擬似コードを図 10 に示す．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 配列を用いて Read と Write 用に 2 つずつ定義されている．そのため，この配列の要素をそれぞれ 1 つずつ増やすことで低優先度処理のキューを追加した．それぞれの配列の 1 番目を Read，2 番目を Write 用に用いているため，3 番目を低優先度処理用のキューとして扱う．

```
1 struct deadline_data {
2   struct rb_root sort_list[3];
3   struct list_head fifo_list[3];
4 }
```

図 10　本機構におけるサブキューの定義

4.3 ディスパッチの制御

3.4 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 つ目の仕組みである，通常の I/O 要求がサブキューに存在しない場合にのみ，ディスパッチ関数を行うための実装の擬似コードを図 11 に示す．`deadline_dispatch_requests` 関数は，ディスパッチを行うキューを決定した後，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 用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 11 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．

```
1  deadline_dispatch_requests(struct deadline_data *dd) {
2    int reads = !list_empty(dd->sort_list[0]);
3    int writes = !list_empty(dd->sort_list[1]);
4    int idles = !list_empty(dd->sort_list[2]);
5    struct request *rq;
6
7    // 優先順序は Read > Write > Idle
8    if (reads) {
9      data_dir = 0;
10     goto dispatch;
11   } else if (writes) {
12     data_dir = 1;
13     goto dispatch;
14   } else if (idle) {
15     data_dir = 2;
16     goto dispatch;
17   }
18   // 3つの並べ替えキューが空の場合は終了
19   return;
20
21 dispatch:
22   rq = sort_list[data_dir];
23   deadline_move_request(rq);
24 }
```

図 11　1 つ目の仕組みに関する擬似コード

2 つ目の仕組みである，一定の時間間隔を空けてディスパッチ関数を行うことで，低優先度処理のスループットを制限するための実装の擬似コードを図 12 に示す．このコードは，図 11 の 14〜17 行目を拡張するものである．この仕組みでは，deadline_data 構造体に対して以下のメンバ変数を定義した．

idle_dispatch_interval
: 低優先度の I/O 要求をディスパッチする時間間隔を保持するためのメンバ変数である．初期値は 200 ms とした．

idle_dispatched_time
: 低優先度の I/O 要求が最後にディスパッチされた時刻を保持するためのメンバ変数である．低優先度の I/O 要求がディスパッチされる度に，jiffies を保存する．

これらの変数を用いた処理の流れを以下に示す．

1. 低優先度の I/O 要求をディスパッチする際に，jiffies と idle_dispatched_time との差分を計算する．差分は，前回のディスパッチからの経過時間を示す．
2. 差分が idle_dispatch_interval を超えていれば，ディスパッチを行う．超えていなければ，次回のディスパッチまで待機をする．
3. ディスパッチを行った場合，jiffies を idle_dispatch_time として保存する．

```
1  deadline_dispatch_requests(struct deadline_data *dd) {
2    ...
3    else if (idles) {
4      // 前回のディスパッチからの経過時間を計算
5      elapsed_time = jiffies - dd->idle_dispatched_time;
6      if (elapsed_time > dd->idle_dispatch_interval) {
7        // idle_dispatched_time を現在時刻に更新
8        dd->idle_dispatched_time = jiffies;
9        data_dir = 2;
10       goto dispatch;
11     }
12   }
13   ...
14 }
```

図 12　2 つ目の仕組みに関する擬似コード

Linux の I/O スケジューラでは，ディスパッチ関数は I/O 要求が追加された時だけ呼び出される．そのため，2 つ目の仕組みによって保留となった I/O 要求は，次のディスパッチまで待機状態となる．他の I/O 要求が追加されない場合，保留中の I/O 要求はいつまでも待機状態となり，やがてタイムアウトエラーを引き起こす．この時の様子を図 13 に示す．さらに，通常の I/O 要求が存在しない状況では，低優先度 の I/O 要求の処理を遅らせてスループットを制限するのも無駄である．

そのため，3 つ目の仕組みの導入し，一定時間後にディスパッチ関数を呼び出し，その後はスループットを制限せずに低優先度の I/O 要求を処理する．ディスパッチ関数を再度呼び出すことで上記の問題を解決する．この仕組みに関する擬似コードを図 14 に示す．`deadline_dispatch_requests` 関数については，図 12 を拡張するものである．`deadline_data` 構造体に対して以下のメンバ変数を定義した．

`idle_redispatch_interval`
: 最後に I/O 要求がディスパッチされてから，再度ディスパッチを行うまでの時間を表すためのメンバ変数．初期値は 500ms とした．

`idle_batch_flag`
: モードのフラグを表すメンバ変数．0 の場合は通常モード，1 の場合はバッチモードを意味する．

`idle_slice_timer`
: カーネルタイマを登録するメンバ変数．

ディスパッチ関数の再呼び出しは，カーネルタイマを用いた．カーネルタイマとは，任意の

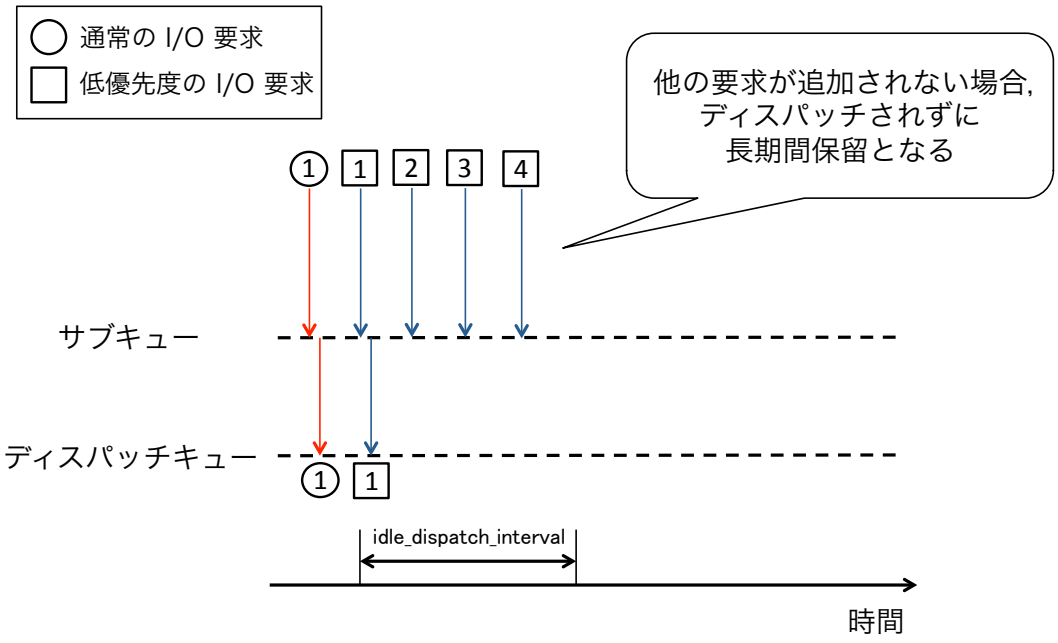


図 13　2 つ目の仕組みにおける問題点

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．カーネルタイマの登録は，deadline_init_queue 関数の中で行う．これにより，I/O スケジューラの登録時にタイマが生成される．カーネルタイマが呼び出す関数として，deadline_redispatch_requests 関数を新たに定義し，タイマに登録した．この関数は，ディスパッチとストラテジルーチンを行う関数である．また，カーネルタイマに対して時刻をセットする関数として deadline_completed_req_fn 関数を新たに定義し，I/O スケジューラ API の elevator_completed_req_fn 関数に登録した．これにより，デバイスドライバが 1 つの I/O 要求を完了する度に，タイマに対して時刻を登録することができる．

モードについては，idle_batch_flag の値で判定する．idle_batch_flag は，deadline_redispatch_requests 関数によってディスパッチされる際に 1 となり，通常の I/O 要求がディスパッチされた場合は 0 となる．この変数が 1 となった後，通常の I/O 要求がディスパッチされるまで，バッチモードが継続するように実装した．

この仕組みに関する処理の流れを以下に示す．

1. I/O 処理が完了すると，直後に deadline_completed_req_fn 関数が呼び出される．この関数は，低優先度 I/O 要求が存在する場合にのみ，タイマに時刻を登録する．登録する時刻は，(jiffies + idle_redispatch_interval) である．この時，登録済み時刻は上書きされる．
2. 現在時刻が登録した時刻になった場合，カーネルタイマによって deadline_redispatch_interval 関数が呼び出される．この関数では，idle_batch_flag を 1 に書き換え，deadline_dispatch_requests 関数を呼び出す．
3. deadline_dispatch_requests 関数では，ディスパッチを行い，直後にストラテジルーチンを行う．
4. ストラテジルーチン後，バッチモードを継続した状態で 1. に戻る．

```
1  /* タイマの生成と関数の登録 */
2  deadline_init_queue(struct deadline_data *dd) {
3    dd->idle_slice_timer.function =
         deadline_redispatch_requests;
4  }
5
6  deadline_completed_request(struct deadline_data *dd) {
7    int idles = !list_empty(sort_list[2]);
8    if (idles)
9      /* タイマに時刻を登録 */
10     mod_timer(&dd->idle_slice_timer,
11         jiffies + dd->idle_redispatch_interval);
12 }
13
14 deadline_redispatch_requests(struct deadline_data *dd) {
15   /* バッチモードへ移行 */
16   idle_batch_flag = 1;
17   deadline_dispatch_requests(dd);
18   ストラテジルーチン呼び出し;
19 }
20
21 deadline_dispatch_requests(struct deadline_data *dd) {
22   ...
23     else if (idles) {
24       if(elapsed_time >= dd->dispatch_interval ||
25         dd->idle_batch_flag) {
26         ...
27       }
28     }
29   ...
30 dispatch:
31   /* 通常の I/O 要求であれば */
32   if (data_dir != IDLE)
33     /* 通常モードへ移行 */
34     dd->idle_batch_flag = 0;
35   ...
36 }
```

図 14　3 つ目の仕組みに関する擬似コード

4.4 期限の設定

既存の実装において，I/O 要求の期限は期限付きキューごとに deadline_data 構造体に定義されている．期限の初期値は Read が 0.5 秒，Write が 5 秒である．そのため，低優先度 I/O 要求の期限を保持する idle_expire を deadline_data 構造体に新たに定義した．低優先度 I/O 要求は，通常の I/O 要求よりも遅延してよい要求であるため，初期値は 10 秒とした．

4.5 I/O 優先度の取得

Linux では，以下の 3 種類の I/O 優先度クラスが定義されている．括弧内の数字は，それぞれの優先度クラスに割り当てられた番号である．

- IOPRIO_CLASS_RT (1)
- IOPRIO_CLASS_BE (2)
- IOPRIO_CLASS_IDLE (3)

優先度クラスには，クラスの IOPRIO_CLASS_RT が最も高く，IOPRIO_CLASS_IDLE が最も低い．また，IOPRIO_CLASS_RT と IOPRIO_CLASS_BE では，クラス内で優先度レベルを 0〜7 まで指定できる．ユーザは，ioprio_set システムコールを呼び出すことで，I/O 要求の優先度を指定することができる．なお，I/O 優先度はと同期書き込み (O_DIRECT, O_SYNC) に対応しており，I/O 優先度は非同期書き込みには対応していない．一般的に I/O 要求の優先度指定は， ionice という Linux コマンドを用いる．図 15 に ionice コマンドの使用してファイルの削除を行う例を示す．ionice コマンドの -c オプションは，優先度クラスを指定するオプションである．ここでは，IOPRIO_CLASS_IDLE を指定している．

```
1 $ ionice -c3 rm -f /tmp/*
```

図 15　ionice コマンドの使用例

実装では，I/O 優先度を取得する関数として req_get_ioprio を用いた．この関数は，request 構造体のメンバ変数 ioprio を参照する関数である．ioprio は，対象プロセスに割り当てるスケジューリングクラスと優先度の両方を表すビットマスクである．ioprio の値を組み立てたり解釈するために，以下のマクロが定義されている．

IOPRIO_PRIO_VALUE(class, data)
: スケジューリングクラス (class) と優先度 (data) を与えると，このマクロは 2 つの値を組み合わせて，ioprio 値を生成し，マクロの結果として返す．

IOPRIO_PRIO_CLASS(mask)

mask (ioprio 値) を与えると，このマクロは I/O クラス要素を返す．I/O クラス要素とは， IOPRIO_CLASS_RT， IOPRIO_CLASS_BE， IOPRIO_CLASS_IDLE のいずれか一つの値である．

IOPRIO_PRIO_DATA(mask)

mask (ioprio 値) を与えると，このマクロは優先度 (data) 要素を返す．

本機構では，最低優先度クラスの I/O 要求であるか，それ以外であるかで処理の方法を分ける．そのため，優先度クラスの解釈するために，IOPRIO_PRIO_CLASS を用いた．マクロによって取得した優先度クラスが，IOPRIO_CLASS_IDLE と一致する場合は低優先度処理を行い，一致しない場合は，通常の処理を行うように実装した．

4.6 設定値のパラメタ化

設定値のパラメタ化は，Linux カーネルが提供する sysfs という機能を用いて実現する．sysfs とは，カーネル空間のリソースをユーザ空間に公開するために用いられる仮想的なファイルシステムである．ユーザは，sysfs 上のファイルの内容を書き換えることで，カーネル内部の設定を動的に変更することができる．Linux カーネルでは，様々な設定が sysfs によってユーザ空間に公開されているが，I/O スケジューラもその 1 つである．例として，I/O スケジューラを Deadline I/O スケジューラに切り替える動作を図 16 に示す．

```
1 # echo deadline > /sys/block/<デバイス名>/queue/scheduler
```

図 16 I/O スケジューラ切り替え時の動作

本機構では，以下の変数をパラメタ化した．

- idle_expire
- idle_dispatch_interval
- idle_redispatch_interval

idle_dispatch_interval と idle_redispatch_interval の値を大きくすることで，低優先度 の I/O 要求が長期間保留されるため，低優先度 の I/O 処理のスループットが抑えることができる．これらの値は，アプリケーションのワークロード，マシンやブロック型デバイスの性能によって適切な値を定める必要がある．また，idle_expire の値を大きくすることで，低優先度 の I/O 要求の期限を延長することができる．この値は，低優先度の I/O 要求が保留される最大時間として値を定める必要がある．

5 評価

本章では，本機構の評価について述べる．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 2 に示す．

OS	CentOS release 6.5 (Final)
Linux カーネル	3.12.7
CPU	Intel (R) Core (TM) i7-3770 CPU @ 3.40GHz 4 コア
メモリ	8 GiB
記憶装置 1	SSD 80 GB (INTEL SSDSA2M080)
記憶装置 2	HDD 320 GB, 7200 rpm (Hitachi HDP72503)

表 2　テスト環境

また，ベンチマークツールは SysBench [3] 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．

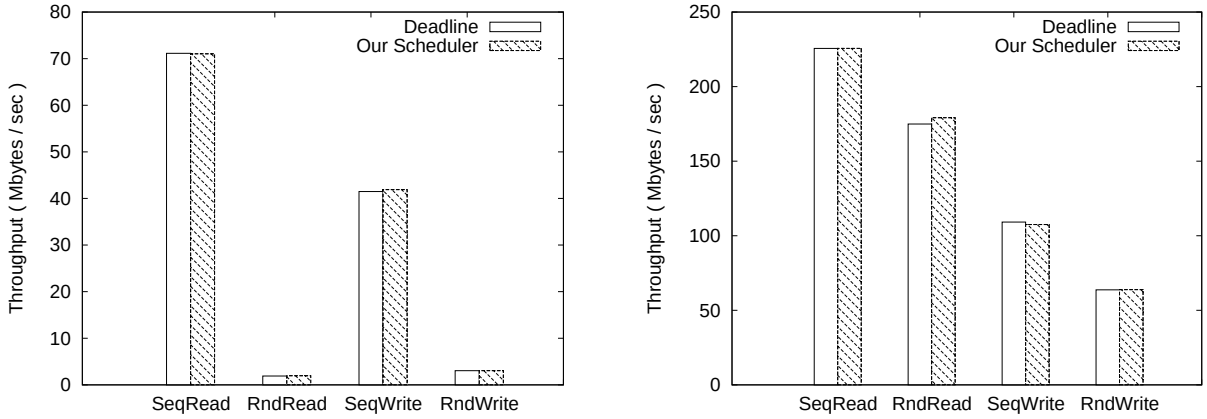
5.1 基本性能の比較

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 17(a)，SSD 上での測定結果を図 17(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 スケジューラと本機構では，性能の差がほぼ同等であることが分かる．



(a) HDD における測定結果　　(b) SSD における測定結果

図 17　基本性能の比較

5.2　低優先度処理のスケジューリング

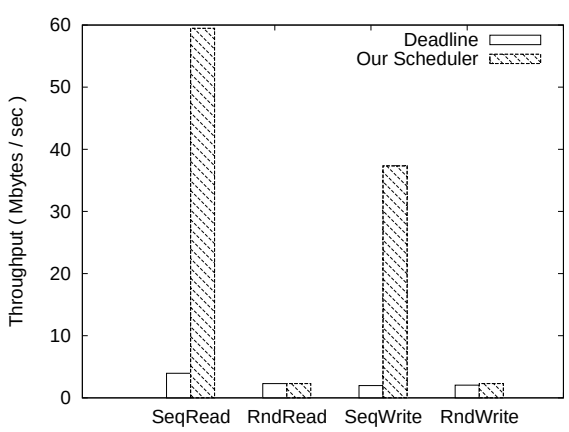
通常の処理と低優先度の処理を同時に実行した場合に，通常の処理がどれだけ処理できるかを比較した．評価は，ファイルアクセスとデータベースアクセスの 2 つのパターンのアクセス方法を用いて行った．

5.2.1　ファイルアクセスによる評価

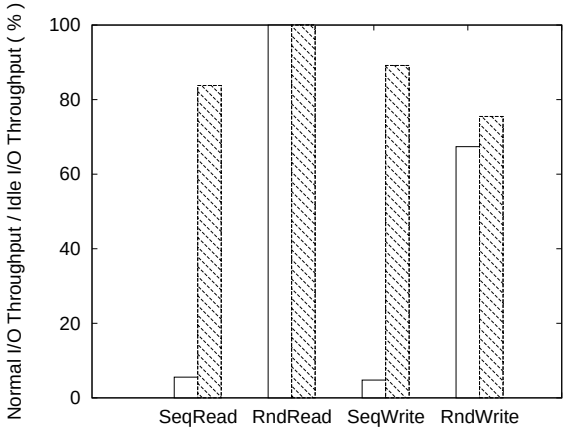
ファイルアクセスによってベンチマークテストを行った．評価の指標は，低優先度処理が行われていない場合のスループットに対する通常の処理のスループットの割合とした．低優先度処理が行われていない場合のスループットは，5.1 節における測定結果を用いた．

ベンチマークテストは，5.1 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て実行した．

HDD 上での測定結果を図 18，SSD 上での測定結果を図 19 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 18(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 18(b) と図 19(b) の結果から，既存の Deadline I/O スケジュー

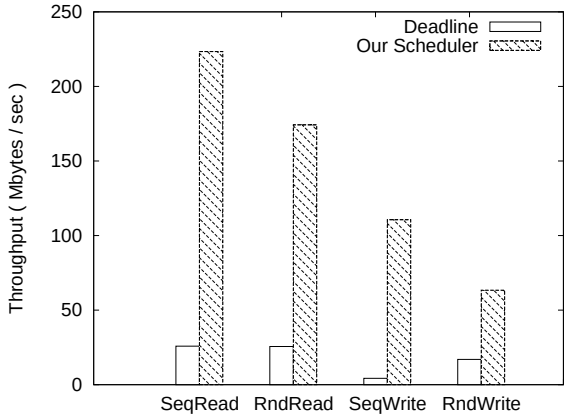


(a) 通常の処理のスループット

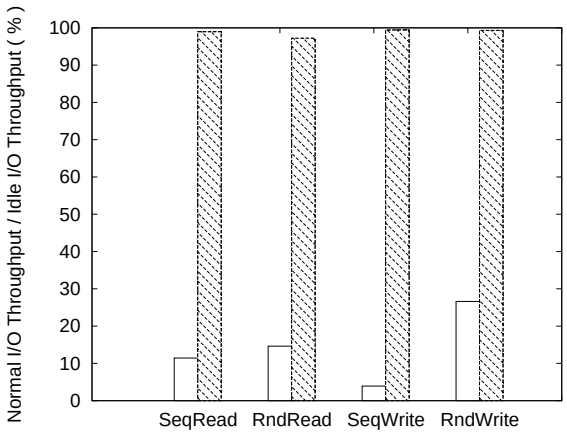


(b) 低優先度処理なしの時に対するスループットの割合

図 18　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 19　SSD における測定結果

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ことが分かる．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とが分かる．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．

5.2.2 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 に対して以下の設定を行った．

`query_cache_size = 0`
: MySQL には，クエリキャッシュという機能がある．これは，クエリの実行結果をメモリにキャッシュするものである．この機能により，キャッシュ後に同じクエリを受け取った時に，クエリを処理する代わりにキャッシュから結果を取り出すことで性能を向上させている．この機能が ON の場合，クエリ実行が行われないため ディスクアクセスも行われない．そのため，今回の実験では値を 0 にすることでクエリキャッシュを OFF にした．

`innodb_buffer_pool_size = 0`
: MySQL では，バッファ機構を搭載している．`innodb_buffer_pool_size` で設定するのは，データベース上のインデックスやデータをキャッシュするためのメモリバッファである．このバッファにより，データ処理をメモリ上で行うことが可能となる．今回は，ディスクアクセスを発生させるために，この設定値を 0 にしてバッファ機構を無効にした．なお，データサイズがメモリサイズを上回る場合は，ディスクアクセスは発生する．

`innodb_flush_method = O_DIRECT`
: このパラメタは，データファイル，ログファイルの読み書き方式を指定するものである．`O_DIRECT` を指定することで，MySQL が行う I/O 処理はダイレクト I/O を用いて行われる．

上記の設定によって，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

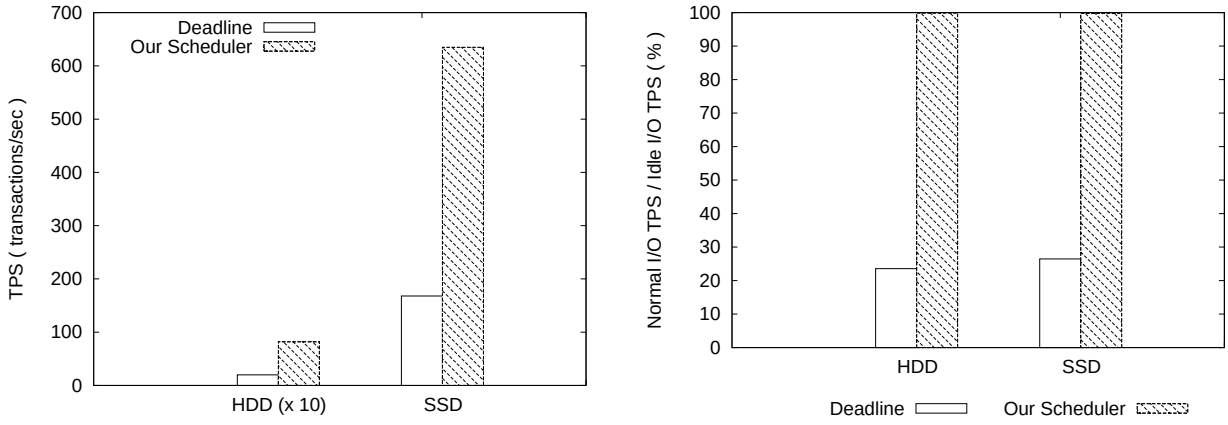
SQL は単純なものにした．1 トランザクション処理中に実行される SQL 文を図 20 に示す．ベンチマークテストでは，1 トランザクション処理中に，図 20 の SQL 文の <FIELD_NAME>, <TABLE_NAME>，および <ID> を適切な値に置き換えたクエリ が 10 回行われる．

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

図 20　1 トランザクション中に実行される SQL

データベース処理と並行して低優先度処理を行った．5.1 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，した場合を比較した結果を図 21 に示す．図 21(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 21(a) には 10 倍した値を載せた．図 21(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，それぞれのグラフにおいて，横軸は記憶装置を表している．図 21(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．



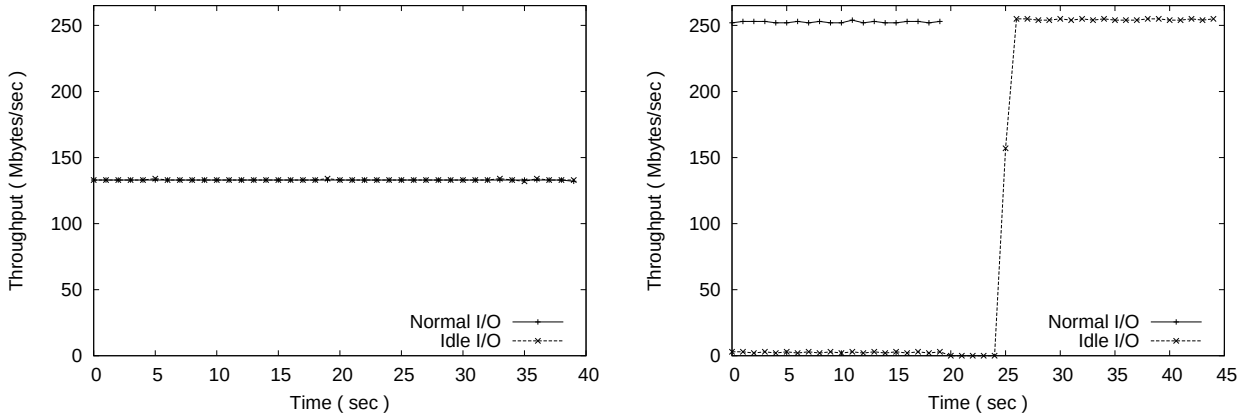
(a) 通常の処理の TPS　　(b) 低優先度処理なしの時の TPS との割合

図 21　SSD における測定結果

5.3 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 `idle_redispatch_interval` 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 22 に示す．図 22 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 22(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ことが分かる．図 22(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ことが分かる．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，`idle_redispatch_interval` 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とが分かる．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．



(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 22　低優先度処理のスループットの様子

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 23 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 23 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 23 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．

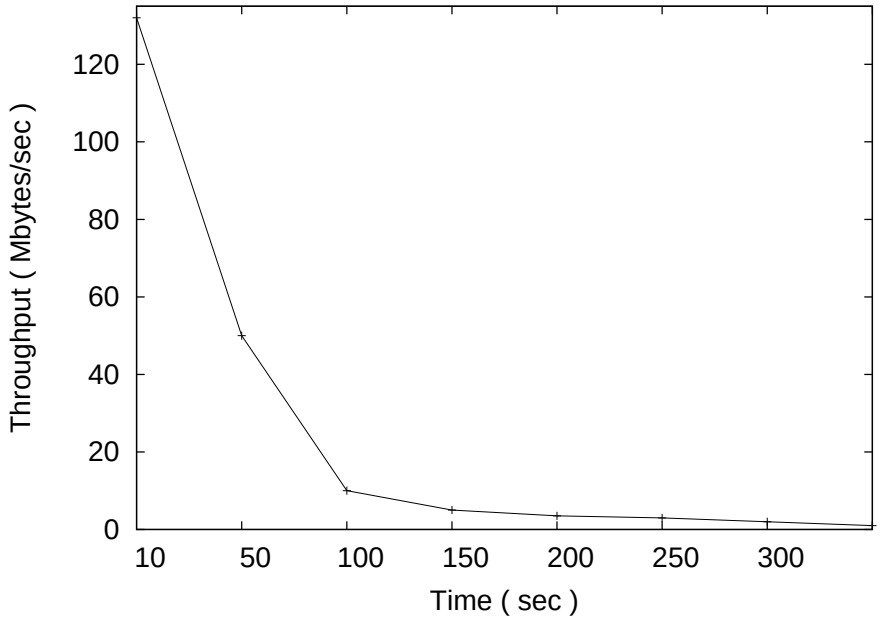


図 23　idle_dispatch_interval と スループットの関係

6 関連研究

Linux カーネルにおける I/O スケジューラ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 [4] や Fusion-io[5] のような高速デバイス向け I/O スケジューラ [6] 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 [7] と Ramon ら [8] 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[9] 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．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，本機構では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．

また，Linux カーネルの機能の 1 つに cgroups (control groups) [10] がある．この機能は，バージョン 2.6.24 以降の Linux カーネルに搭載された機能で，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・隔離するものである．cgroups を用いることで，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．しかし，プロセスごとに設定が必要であるため煩雑である．また，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．本研究は，LKM を用いて実装したため導入コストが低く，`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，3 章で提案した 3 つ目の仕組みにより，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，マシンのスループットを有効活用して処理を行う．

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．

既存の Deadline I/O スケジューラを用いたシステムでオンラインメンテナンスを行った場合は，メンテナンス処理の I/O 負荷によって，データベース処理の性能を著しく低下させる可能性がある．本研究における評価では，TPS が 通常時の 30% 以下になるという結果が得られた．このような状況が，実際の Web サービス上で発生した場合，サイトの閲覧ができない，通信タイムアウトにより決済が中断されるなどの問題が多数発生することとなる．このように，オンラインメンテナンスは機会損失のトリガーになりうる大変危険な作業である．本研究では，オンラインメンテナンスによるサービスへの影響を及ぼさないようにする方法の 1 つとして，I/O バウンドな処理に着目し，本機構を提案した．本機構の導入により，オンラインメンテナンスによる負荷を最小限に抑えることが可能となる．ただし，本機構がカバーしているのは I/O 負荷のみであることに注意する必要がある．CPU バウンドなメンテナンス処理については，別の対策が必要である．また，CPU 負荷のケアができた場合もその他の問題はいくつも存在する．このように，安全にメンテナンス処理を行うには多くの問題に対する対策を行う必要がある．そのため，オンラインメンテナンスを行う場合は常にサービスを最優先に考え，メンテナンス処理の影響範囲を想定し，適切な対策を講じることが大切である．

本論文で提案した機構は，低優先度の優先度クラス (`IOPRIO_CLASS_IDLE`) には対応しているが，その他の 2 つの優先度クラス (`IOPRIO_CLASS`，`IOPRIO_RT`) には対応していないため改善が必要である．また，優先度クラスの指定を検出することにより低優先度処理の判定を行っているが，この実装には改善点がある．優先度指定には，優先度クラスの他に優先度レベルを指定することができる．そのため，優先度レベルによって優先順位を決定するようなアルゴリズムにすることで，低優先度クラスの中において，優先度を 8 段階で指定することが可能となる．優先度レベルの値と `idle_dispatch_interval` の値を対応付けることで，より柔軟なシステムになることが考えられる．また，今回は特定のカーネルのバージョンに対して実装を行ったが，汎用性を上げるためには，long term のカーネルに対しても実装する必要がある．

謝辞

本研究を行うにあたり，終始変わらぬご指導を賜りました鵜川始陽助教，岩崎英哉教授並びに中野圭介准教授に深く感謝いたします．また，研究を進めるにあたって熱心なご指導を頂きました岩崎研究室，中野研究室，鵜川研究室の皆様に心から御礼申し上げます．

参考文献

[1] Oracle, 3.15 Linux でのディスク I/O スケジューラの設定, http://docs.oracle.com/cd/E49329_01/install.121/b71312/pre_install.htm.

[2] DRBD, チューニングの推奨事項, http://www.drbd.jp/users-guide/s-latency-tuning.html.

[3] Kopytov, A.: SysBench manual (2009), http://sysbench.sourceforge.net/docs/.

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

[5] Fusion-io, http://www.fusionio.com/.

[6] Dong, R.: TPPS: Tiny Parallel Proportion Scheduler (2013), https://lkml.org/lkml/2013/6/4/949.

[7] Seelam, S. and Romero, R.: Enhancements to Linux I/O Scheduling (2005).

[8] Nou, R. and Giralt, J.: Automatic I/O scheduler selection through online workload analysis (2010), IEEE.

[9] Lunde, C. H.: Improving Disk I/O Performance on Linux, Master's thesis, Hvard Espeland (2009), http://agesage.co.jp/.

[10] Menage, P.: CGROUPS: https://www.kernel.org/doc/Documentation/cgroups/cgroups.txt.

[11] Bovet, D. P. and Cesati, M.: 詳解 Linux カーネル 第 3 版, オライリー・ジャパン (2007).

[12] 高橋浩和, 小田逸郎, 山幡為佐久：Linux カーネル 2.6 解読室, ソフトバンク クリエイティブ (2006).

[13] Axboe, J.: Notes on the Generic Block Layer Rewrite in Linux 2.5 (2002), https://www.kernel.org/doc/Documentation/block/biodoc.txt.