2006－OS－101（10）
2006／2／17

ディレクトリに着目したバッファキャッシュ制御法の実現

齊藤 圭　　乃村 能成　　谷口 秀夫
岡山大学大学院自然科学研究科

既存の多くのオペレーティングシステムでは，I/O バッファキャッシュをブロック単位で LRU 管理している．一方，応用プログラムは，ファイル単位で扱う．このため，ファイルアクセスの観点から I/O バッファキャッシュを管理することが考えられる．そこで，本稿では，ブロックとファイルの対応関係を考慮して I/O バッファキャッシュを管理し，一連の仕事で発生するディスクアクセスは同一ディレクトリ内のファイル群に集中するのではないかという考えに基づいて，特定ディレクトリ下のファイルを優先的にキャッシュする I/O バッファキャッシュ制御法を提案する．さらに，提案方式の実装，および，評価について述べる．

Directory oriented buffer cache mechanism

Kei SAITOU　Yoshinari NOMURA　Hideo TANIGUCHI
The Graduate School of Natural Science and Technology,Okayama University

Average operating systems manage disk I/O cache by a block unit with LRU method. On the other hand, Application programs (APs) handle data from a viewpoint of file unit. To fill up the gap, we propose a directory oriented buffer cache mechanism that reflects on the demands of APs. Our mechanism is based on a thought that a disk access pattern occurred from a series of job will be settled in a corresponding particular directory. We give a high priority to the important directory, which is associated with an important job. This paper describes an implementation and an evaluation of our method.

1　はじめに

ディスク装置のバッファキャッシュ管理は，オペレーティングシステム (以降，OS) の重要な仕事である．これを実現するアルゴリズムをブロック置き換えアルゴリズムと呼ぶ．

ブロック置き換えアルゴリズムついて，現在までに様々な研究 [1]〜[8] が行われてきた．ブロック置き換えアルゴリズムは，大きく 3 つに分類される．1 つ目は，各ブロックの参照順序や参照頻度に基づいてブロック置き換えを行う方式である．この方式の代表的な例として，LRU，FIFO，LFU，FBR[1]，LRU-k[2]，IRG[3] が存在する．2 つ目は，ユーザによってあらかじめ提供されたブロック参照パターンに基づいてブロック置き換えを行う方式である．この方式の代表的な例として，ACFC[4]，UBM[5] が存在する．3 つ目は，ブロック参照の定期性に基づいてブロック置き換えを行う方式である．この方式の代表例として，2Q[6]，SEQ[7]，EELRU[8] が存在する．従来の OS では，バッファキャッシュをブロック単位で LRU 管理することが一般的である．処理コストが低く，一般的利用状況において，よく機能するからである．また，これらの方式は全て，ブロックアクセスに着目した方式であるため，ファイルは意識していない．

一方，応用プログラム (以降，AP) は，ブロックではなく，ファイルという抽象度の一段高いレベルでディスクを扱う．そのため，ユーザは，ファイルアクセスの観点からキャッシュ管理を発想することが自然である．例えば，今から行う特定の仕事に必要な，特定のファイル群を優先的にキャッシュしたいといった要求である．

しかしながら，この要求を，従来の LRU 方式を適用している OS に提示することは難しい．つまり，ユーザの仕事別の事情をうまくキャッシュアルゴリ

ズムに反映させることが難しいのである．例えば，ある重要な仕事を実行中に，その仕事と並行して重要でない仕事を実行しているとする．従来の LRU 方式では，重要でない仕事に使用したファイル群によって，重要な仕事に使用したファイル群に関連するブロックがバッファキャッシュの中から削除されてしまうという場合がある．この場合，重要な仕事が，以前に使用したファイルを再度使用する際に再度ディスク I/O を行わなければならない．もし，重要な仕事に必要なファイル群に関連するブロックをそのままバッファキャッシュ内に残すことができるならば，無駄なディスク I/O の削減が期待できる．

そこで，本稿では，新たなブロック置き換えアルゴリズムとして，ブロックとファイルの対応関係を考慮してバッファキャッシュを管理し，ある一連の仕事で発生するディスクアクセスは同一ディレクトリ内のファイル群に集中するのではないかという考えに基づいて，指定ディレクトリ下のファイルを優先的にキャッシュするバッファキャッシュ制御法を提案する．この提案方式は，ある重要な仕事に使用するファイル群が存在するディレクトリを指定することにより，その仕事に使用するファイル群を優先的にキャッシュさせる．これにより，他の仕事を並行して行っている場合でも，重要な仕事にはあまり影響を与えずに処理することができる．

2　ディレクトリ優先方式

2.1　基本方針

従来の OS では，読み込んだブロックが持つバッファをバッファプールに蓄え，バッファプールの空きが無くなると，最も古くに参照されたブロックから置き換えられる LRU 方式を一般的に利用している．ここでは，特定ディレクトリ下のファイルを優先的にキャッシュすることに着目したバッファキャッシュ制御法を提案する．以降，この方式をディレクトリ優先方式と呼ぶ．

例えば，図 1 に示すように，仕事 A と仕事 B が存在するとする．図 1 において，directory_A は仕事 A に使用されるファイルが置かれているディレクトリを表し，directory_B は仕事 B に使用されるファイルが置かれているディレクトリを表し，それぞれのファイルのサイズは全て 1 ブロックとする．また，仕事 A は重要な仕事であり，仕事 B はあまり重要な仕事ではないとする．ここで，仕事 A と仕事 B を並行して行うとすると，ユーザは，重要な仕事 A を先に終わらせたいという要求を抱くことが一般的である．

この例において，file1，file3，file2，file4 の順に繰り返しファイルを読み込む場合，従来の LRU 方式とディレクトリ優先方式との 2 つの方式で比較したものを図 2 に示す．また，バッファキャッシュが管理できるブロック数は 3 個とし，これらをキャッシュ領域という領域で管理する．図中の四角はファイルに関連するブロックを表し，四角中の文字はファイル名を示している．図 2 の従来の LRU 方式において，file2 まで読み込んだ時点のキャッシュ領域を (1) とし，実線で囲んでいる．また，file4 まで読み込んだ時点のキャッシュ領域を (2) とし，破線で囲んでいる．(1) の時点ではキャッシュ領域中に file1，file2，file3 が確保されている．しかし，(2) の時点で，キャッシュ領域中から file1 のブロックが削除されているため，その後の file1 の読み込みはディスク I/O を行わなくてはならない．

そこで，本論文で提案するディレクトリ優先方式は，ディレクトリに着目する．重要な仕事である仕事 A に使用するファイル (file1，file2) が存在するディレクトリ directory_A を特定ディレクトリとして指定する．そうすると，directory_A 下のファイルを優先的にキャッシュすることができる．図 2 において，従来の LRU 方式と同様のファイル読み込みが発生した場合，directory_A 下に置かれているファイルである file1，file2 に関連するブロックを保護領域で管理する．そして，file3，file4 に関連するブロックについては，残りのキャッシュ領域で管理する．これにより，file1，file2 に関連するブロックはキャッシュ領域とは別の領域に確保されているため，保護領域から解放されない限り，ディスク I/O は発生しない．

このようにバッファキャッシュ制御を行うことにより，ディスク I/O の回数の削減が見込める．重要な仕事 A に使用するファイル群に関するディスク I/O の回数を削減することにより，仕事 A を先に終了させることができる．よって，重要な仕事 A を先に終わらせたいというユーザの要求を満たすことができる．

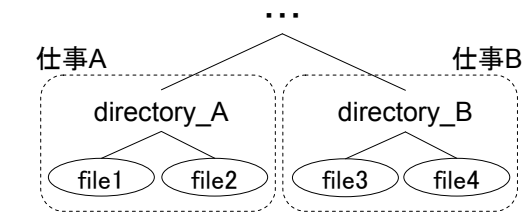

図 1 ある仕事の一例

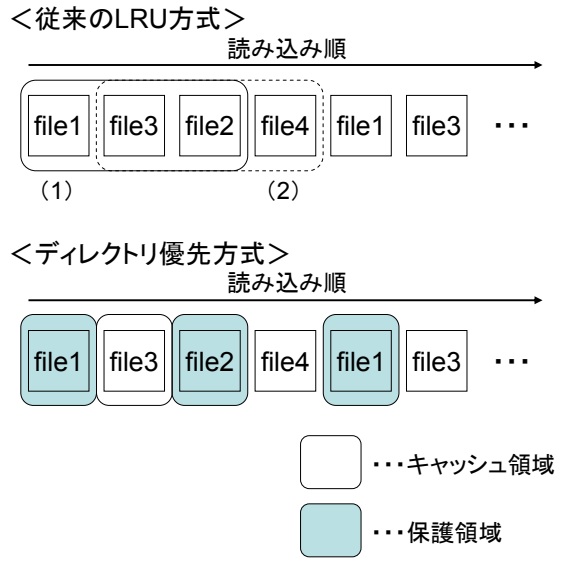

図 2 LRU 方式とディレクトリ優先方式の比較

2.2 機能概要

ディレクトリ優先方式の概要図を図 3 に示し，以下に説明する．

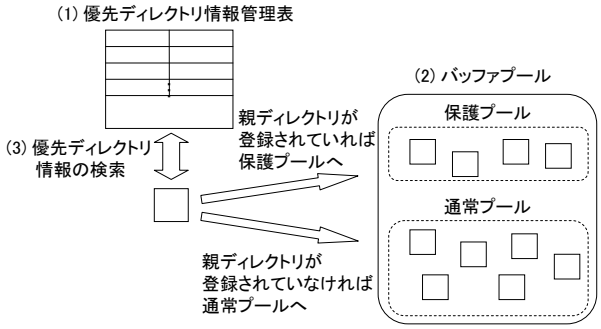

図 3 ディレクトリ優先方式概要

(1) 優先ディレクトリ情報の管理
指定したディレクトリ (以降，優先ディレクトリ) の情報を管理する管理表 (以降，優先ディレクトリ情報管理表) を持ち，バッファを割り当てる際の検索に使用する．

(2) バッファプールの分割
バッファプールは，通常プールと保護プールで構成されている．通常プールでは，親ディレクトリの情報が優先ディレクトリ情報管理表に登録されていないファイルに関連するブロックを管理する．一方，保護プールでは，親ディレクトリの情報が優先ディレクトリ情報管理表に登録されているファイルに関連するブロックを管理する．

(3) ブロック解放の判定
バッファキャッシュは，ブロック解放手続きの際に，読み込んだブロックに対応するファイルの親ディレクトリの情報が優先ディレクトリ情報管理表に登録されているかどうかを検索する．その検索で得られた判定に基づいて，以下の処理を行う．

(a) 親ディレクトリの情報が優先ディレクトリ情報管理表に登録されているファイルに関するブロックが持つバッファを保護プールへ移す．

(b) 親ディレクトリの情報が優先ディレクトリ情報管理表に登録されていないファイルに関連するブロックが持つバッファを通常プールへ移す．

ディレクトリ優先方式では，バッファプールが管理するブロックを優先ディレクトリ情報管理表の情報に基づいて，通常プールと保護プールのどちらか一方に移す．ブロック置き換え時には，通常プールのブロックから優先して置き換えられる．保護プールにあるブロックは，通常プールにあるブロックが全て無くならない限り，置き換えられることはない．以降，通常プールと保護プールを合わせたものをバッファプールと呼ぶ．

3 実現方式

3.1 優先ディレクトリ情報管理表

OS 内に，優先ディレクトリの情報を管理する管理表を作成した．優先ディレクトリ情報管理表では，優先的にキャッシュしたいファイル群が存在するディレクトリの情報を管理している．優先ディレクトリ情報管理表で管理する情報は，ディレクトリのパス名と，そのディレクトリの i ノード番号である．この 2 つの情報をまとめて優先ディレクトリ情報と呼ぶ．

優先ディレクトリ情報管理表に優先ディレクトリ情報を追加することにより，優先ディレクトリ下のファイルを優先的にキャッシュするように指定する．

3.2 バッファプール

バッファプールでは，各バッファがそれぞれのリストに繋がれている．それぞれ，LOCKED リストはロックされたバッファ，EMPTY リストは空バッファ，EMPTYKVA リストは仮想アドレスを割り当てられた空バッファ，DIRTY リストは現在ダーティであるバッファ，AGE リストは先読みされたバッファ，CLEAN リストは再利用可能なバッファを保持する．

例えば FreeBSD の場合，新規にバッファを要求されると EMPTY リストのバッファを使用し，使用済みのバッファを CLEAN リストの最後尾に繋ぐ．もし EMPTY リストにバッファが存在しなければ，CLEAN リストの先頭のバッファを再利用する．また，バッファプール内に存在するブロックが要求された場合，そのバッファからデータを読み込み，そのバッファは CLEAN リストの最後尾に繋ぎ変えられる．

図 4 に示すように，ディレクトリ優先方式では，従来使用されているバッファプールに，新たに DIR リストを追加した．DIR リストに繋がれたバッファを保護プールのバッファとして扱う．また，DIR リスト以外のリスト，つまり，従来のバッファプールで使用されているリストに繋がれたバッファは，全て通常プールのバッファとして扱う．

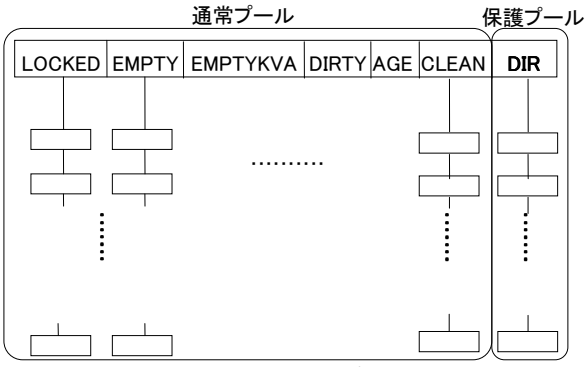

図 4 バッファプール

3.3 処理の流れ

本方式は，優先ディレクトリ情報管理表更新処理，バッファ解放処理，バッファ割り当て処理の 3 つの処理から成る．以下に，各処理の概要を説明する．

(1) 優先ディレクトリ情報管理表更新処理
優先ディレクトリ情報を優先ディレクトリ情報管理表に追加または削除する．

(2) バッファ割り当て処理
優先ディレクトリ情報管理表を参照して，読み込んだブロックに対応するファイルの親ディレクトリ情報が登録されているかどうか検索する．その判定に基づいて，そのブロックが持つバッファを保護プールまたは通常プールへ移す．

(3) バッファ解放処理
新規のブロック読み込みが発生した時に，通常プール中の最も古くに参照されたバッファを解放する．その際に通常プール中にバッファが存在しない場合は，保護プール中の最も古くに参照されたバッファを解放する．

上記の 3 つの処理について，以降で詳しく説明する．

3.4 優先ディレクトリ情報管理表更新処理

優先ディレクトリ情報管理表更新処理は，以下の 2 つの処理から成る．

(1) 優先ディレクトリ情報追加処理
(2) 優先ディレクトリ情報削除処理

これらの処理は，ユーザによって任意の契機で行われる．

優先ディレクトリ情報追加処理

優先ディレクトリ情報追加処理は，優先ディレクトリとして指定したいディレクトリの情報を，優先ディレクトリ情報管理表に追加する．この処理を実現するアルゴリズムは，以下の通りである．

(1) ユーザから，優先ディレクトリとして指定するディレクトリのパス名を受け取る．
(2) 受け取ったパス名から，そのディレクトリの i ノード番号を取得する．
(3) 取得した i ノード番号から，そのディレクトリが既に優先ディレクトリ情報管理表に登録されているかどうか検索する．
(4) そのディレクトリが登録されていなければ，そのディレクトリのパス名と i ノード番号を優先ディレクトリ情報管理表に追加する．

優先ディレクトリ情報削除処理

優先ディレクトリ情報削除処理は，優先ディレクトリとして指定されているディレクトリの情報を優先ディレクトリ情報管理表から削除する．この処理を実現するアルゴリズムは，以下の通りである．

(1) ユーザから，削除したい優先ディレクトリのパス名を受け取る．

(2) 受け取ったパス名から，そのディレクトリのiノード番号を取得する．
(3) 取得したiノード番号から，そのディレクトリが優先ディレクトリ情報管理表に登録されているかどうか検索する．
(4) そのディレクトリが登録されていれば，そのディレクトリのパス名とiノード番号を優先ディレクトリ情報管理表から削除する．

3.5 バッファ割り当て処理

バッファ割り当て処理では，優先ディレクトリ情報管理表を参照し，読み込んだブロックに対応するファイルの親ディレクトリが登録されているかどうか判定する．その判定に基づいて，読み込んだブロックが持つバッファを保護プールまたは通常プールのどちらかのリストに繋げる．この処理を実現するアルゴリズムは，以下の通りである．

(1) 読み込んだブロックに対応するファイルの親ディレクトリのiノード番号を取得する．
(2) 取得したiノード番号が，優先ディレクトリ情報管理表の中に登録されているかどうかを検索する．
  (a) 登録されていた場合
  読み込んだブロックが持つバッファを保護プールのリストに繋げる．
  (b) 登録されていない場合
  読み込んだブロックが持つバッファを通常プールのリストに繋げる．

3.6 バッファ解放処理

バッファ解放処理は，新規のブロック読み込みが発生した時に，不要なバッファをバッファプールから削除する．この処理を実現するアルゴリズムは，以下の通りである．

(1) 通常プールにバッファが存在する場合
  通常プール中の最も古くに参照されたバッファをバッファプールから削除する．
(2) 通常プールにバッファが存在しない場合
  保護プール中の最も古くに参照されたバッファをバッファプールから削除する．

4 評価と考察

4.1 環境

ディレクトリ優先方式をFreeBSD 4.3-RELEASEに実装した．評価環境を以下に示す．

(1) 使用OS：FreeBSD 4.3-RELEASE
(2) CPU：Intel Pentium Ⅲ (800 MHz)
(3) バッファキャッシュサイズ：4 MB
  なお，システム維持のために常時確保されるスペースが存在するため，実際使用できるスペースは約3.3 MB強である．

また，FreeBSD 4.3-RELEASEでは，バッファキャッシュ内に該当するブロックが見つからなかった場合，実メモリ上に該当するページがあるかどうかを探索する機能が実装されている．バッファキャッシュで行われるバッファ制御のみで実行時間とディスクI/O回数の測定を行うために，この機能を停止させて測定を行った．

4.2 事例評価

以下の3つの事例について測定実験を行った．

(事例1) 連続的なファイル読み込み実験
  この実験では，与えられた読み込み順に従って連続的にファイル読み込みを行い，その処理にかかる時間とディスクI/Oの回数を測定する．この実験は，ディレクトリ優先方式の動作確認として，ディレクトリ優先方式に最も都合の良いと思われる事例を想定している．
(事例2) カーネルmake実験
  この実験では，カーネルmakeを行い，その処理にかかる時間とディスクI/Oの回数を測定する．この実験は，APの一例としてカーネルmake処理を取り上げ，ディレクトリ優先方式が実際のAPで効果があるかどうかを検証する．
(事例3) 負荷を与えた状態でのカーネルmake実験
  この実験では，測定を行う計算機に別のディスクI/O負荷をかけて，その状態でカーネルmakeを行い，その処理にかかる時間とディスクI/Oの回数を測定する．この実験は，ユーザが，カーネルmake実行中の待ち時間にWebページを閲覧して時間を潰すといった事例を想定している．

以下に，それぞれの事例について詳しく説明する．

事例1

この実験は，与えられた読み込み順に従って連続的にファイルを読み込み，その実行時間とディスクI/Oの回数を測定する．従来のLRU方式とディレクトリ優先方式とでそれぞれ実験を行った．実験の手順を以下に示す．

(1) テストファイルを6つ(file1, file2, file3, file4, file5, file6)用意し，図5のように配置する．

これらのテストファイルのサイズは，それぞれ 1 MB とする．

(2) 図 5 において，directory2 下のファイルを優先的にキャッシュするように指定する．

(3) file1〜file6 の順でファイルの読み込みを行う．ファイルの読み込みは，1024 B 単位で read システムコールを繰り返し発行することで行った．これを 1 セットとし，計 10 セットの読み込みを行う．

(4) 上記 (3) の操作を 10 回行い，それぞれ実行時間とディスク I/O の回数を測定し，平均値を算出する．

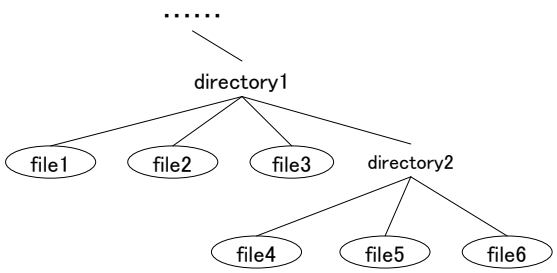

図 5 テストファイルの位置関係

事例 2

この実験は，カーネル make を行い，実行時間とディスク I/O の回数を測定する．従来の LRU 方式とディレクトリ優先方式とでそれぞれ実験を行った．実験の手順を以下に示す．

(1) カーネル make に頻繁に参照されるファイルを優先的にキャッシュするよう指定する．具体的には，カーネル make に頻繁に参照されるヘッダファイル群が存在するディレクトリを指定する．

(2) カーネル make を実行する．

(3) 上記 (2) の操作を 3 回行い，それぞれの実行時間とディスク I/O の回数を測定し，平均値を算出する．

なお，この実験において，優先ディレクトリを指定しない場合についても測定を行った．

事例 3

この実験は，計算機に別のディスク I/O 負荷をかけた状態でカーネル make を行い，この時のカーネル make の実行時間とディスク I/O の回数を測定する．従来の LRU 方式とディレクトリ優先方式とでそれぞれ実験を行った．実験の手順を以下に示す．

(1) カーネル make に全く関係無いテストファイルを 5 つ (file1，file2，file3，file4，file5) 用意する．これらのテストファイルのサイズは，それぞれ 1 MB とする．

(2) file1〜file5 の順でファイルの読み込みを行う．ファイルの読み込みは，1024 B 単位で read システムコールを繰り返し発行することで行った．このファイル読み込み処理を繰り返し行うことにより，計算機に常に一定のディスク I/O 負荷を与える．

(3) カーネル make に頻繁に参照されるファイルを優先的にキャッシュするよう指定する．具体的には，カーネル make に頻繁に参照されるヘッダファイル群が存在するディレクトリを指定する．

(4) カーネル make を実行する．

(5) 上記 (4) の操作を 3 回行い，それぞれの実行時間とディスク I/O の回数を測定し，平均値を算出する．

なお，優先ディレクトリとして指定するディレクトリは，事例 2 と同じディレクトリとした．また，事例 2 と同様に，優先ディレクトリを指定しない場合についても測定を行った．

4.3 結果と考察

事例 1

事例 1 の結果を表 1 に示す．表 1 から，ディレクトリ優先方式は，従来の LRU 方式に比べて，実行時間は約 29.9%減少し，ディスク I/O の回数は約 39.5%減少した．これは以下の理由による．

バッファキャッシュサイズ 4 MB に対してテストファイルは計 6 MB である．このため，従来の LRU 方式では，2 セット目以降の読み込みの際，読み込もうとするファイルに関連するブロックが持つバッファはキャッシュ内に既に存在しないので，読み込み時に毎回ディスク I/O が発生する．一方，ディレクトリ優先方式では，directory2 下のファイルを優先的にキャッシュするよう指定した．このため，file4，file5，file6 に関連するブロックが持つバッファは，2 セット目以降は常にバッファキャッシュ内に存在し続け，残りのバッファを用いて file1，file2，file3 の読み込みを行う．これにより，実行時間とディスク I/O の回数の削減が見られたと考えられる．

この結果より，ディレクトリ優先方式は正常に動作していることを確認した．

表 1 事例 1 の実験結果

	実行時間 [s]	ディスク I/O[回]
従来方式	7.18	8251
提案方式	5.03	4991

事例 2

事例 2 の実験結果において，優先ディレクトリを指定した場合の結果について表 2 に，優先ディレクトリを指定しない場合の結果について表 3 に示す．

表 2 から，ディレクトリ優先方式は，従来の LRU 方式に比べて，実行時間は約 1.4%減少し，ディスク I/O の回数は約 5.2%減少した．これは以下の理由による．

カーネル make に頻繁に使用されるヘッダファイル群に関連するブロックを優先的にキャッシュすることにより，これらのファイル群に関連するブロックはバッファキャッシュ内に存在し続ける．そして，残りのバッファを用いて，その他のファイルの読み込みを行う．これにより，実行時間とディスク I/O の削減が見られたと考えられる．

また，表 3 から，ディレクトリ優先方式は，従来の LRU 方式に比べて，実行時間は約 0.1%増加し，ディスク I/O の回数は約 0.2%増加した．これは以下の理由による．

優先ディレクトリ情報管理表には，優先ディレクトリ情報が全く登録されていない．これより，保護プールは使用されず，通常プールのみを使用する．つまり，優先ディレクトリを指定していない場合は，従来の LRU 方式と同等のバッファ制御を行うことになる．しかし，ディレクトリ優先方式では，ファイルの読み込みを行う際に優先ディレクトリ情報管理表を検索するかどうかの判定処理が行われる．ファイルの読み込みを行った際に毎回この判定処理が行われるため，この処理の分だけ実行時間とディスク I/O の回数が増加したものと考えられる．この増加分は，カーネル make 処理においてディレクトリ優先方式の基本オーバヘッドと見なすことができる．

表 2 事例 2 の実験結果 (優先ディレクトリ有り)

	実行時間 [s]	ディスク I/O[回]
従来方式	456.74	13494
提案方式	450.12	12794

表 3 事例 2 の実験結果 (優先ディレクトリ無し)

	実行時間 [s]	ディスク I/O[回]
従来方式	456.74	13494
提案方式	457.10	13523

事例 3

事例 3 の実験において，優先ディレクトリを指定した場合の結果について表 4 に，優先ディレクトリを指定しない場合の結果について表 5 に示す．

表 4 から，ディレクトリ優先方式は，従来の LRU 方式に比べて，実行時間は約 32.9%減少し，ディスク I/O の回数は約 42.7%減少した．これは以下の理由による．

この実験では，カーネル make に全く関係無いファイルの読み込みを並行して行っている．このため，従来の LRU 方式では，カーネル make に頻繁に使用されるファイル群に関連するブロックがバッファキャッシュ内から削除されてしまう．一方，ディレクトリ優先方式では，カーネル make に頻繁に使用されるヘッダファイル群に関連するブロックを優先的にキャッシュすることにより，これらのファイル群に関連するブロックはバッファキャッシュ内に存在し続ける．そして，残りのバッファを用いて，その他のファイルの読み込みを行う．これにより，実行時間とディスク I/O の削減が見られたと考えられる．

また，表 5 から，ディレクトリ優先方式は，従来の LRU 方式に比べて，実行時間は約 0.4%増加し，ディスク I/O の回数は約 0.7%増加した．これは以下の理由による．

事例 2 の優先ディレクトリ無しと同様に，ディレクトリ優先方式の基本オーバヘッドの分だけ，実行時間とディスク I/O の回数が増加したものと考えられる．

表 4 事例 3 の実験結果 (優先ディレクトリ有り)

	実行時間 [s]	ディスク I/O[回]
従来方式	1675	43114
提案方式	1124	24698

表 5 事例 3 の実験結果 (優先ディレクトリ無し)

	実行時間 [s]	ディスク I/O[回]
従来方式	1675	43114
提案方式	1682	43396

5 おわりに

本論文では，バッファキャッシュのブロック置き換えアルゴリズムにおいて，指定されたディレクトリ下のファイルを優先的にキャッシュすることに着目したディレクトリ優先方式を提案した．

バッファプールを指定されたディレクトリ下のファイルに関連するブロックを蓄える保護プールと，その他のファイルに関連するブロックを蓄える通常プールの 2 つに分割した．これにより，指定されたディレクトリ下のファイルを優先的にキャッシュすることができるようになった．

ディレクトリ優先方式を実装し，また，3 つの事例について事例評価を行った．事例 1 において，ディレクトリ優先方式は，従来の LRU 方式に比べて，実行時間は約 29.9%減少し，ディスク I/O の回数は約 39.5%減少することを明らかにした．この結果より，ディレクトリ優先方式は正常に動作していることが確認できたことを示している．事例 2 において，ディレクトリ優先方式は，従来の LRU 方式に比べて，実行時間は最大約 1.4%減少し，ディスク I/O の回数は最大約 5.2%減少することを明らかにした．これは，ディレクトリ優先方式は，カーネル make 処理においても効果を発揮することを示している．そして，事例 3 において，ディレクトリ優先方式は，従来の LRU 方式に比べて，実行時間は最大約 32.9%減少し，ディスク I/O の回数は最大約 42.7%減少することを明らかにした．これは，ある重要な仕事に関連するファイル群を優先的にキャッシュしたいといったユーザの要求を満たすことができたことを示している．

今後の課題として，より一般的な事例についての評価を行い，ディレクトリ優先方式の有効性を検討することが挙げられる．

参考文献

[1] J. T. Robinson and M. V. Devarakonda, “Cache Management Using Frequenfcy-Based Repalacement”, Proc. the 1990 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems, pp. 134-142(1990)

[2] E. J. O’Neil, P. E. O’Neil, and G. Weikum, “The LRU-k Page Replacement Algorithm for Database Disk Buffering”, Proc. the 1993 ACM SIGMOD Conference, pp. 297-306(1993)

[3] V. Phalke and B. Gopinath, “An Inter-Reference Gap Model for Temporal Locality in Program Behavior”, Proc. the USENIX Summer 1994 Technical Conference, pp. 291-300(1995)

[4] P.Cao , E. W. Falten. and K. Li, “Application Controlled File Caching Policies”, Proc. the USENIX Summer 1994 Technical Conference, pp. 171-182(1994)

[5] J. M. Kim, J. Choi, J. Kim, and Sam H. Noh, “A Low-Overhead High-Peformance Unified Buffer Management Scheme that Exploits Sequential and Looping References”, Proc. OSDI 2000, pp. 119-134(2000)

[6] T. Johnson and D. Shasha, “A Low Overhead High Performance Buffer Management Replacement Algorithm”, Proc. the 20th International Conference on VLDB, pp. 297-306 (1993)

[7] G. Glass and P. Cao, “Adaptive Page Replacement Based on Memory Reference Behavior”, Proc. the 1997 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems, pp. 115-226(1997)

[8] Y. Smaragdakis, S. Kaplan, and P. Wilson, “Simple and Effective Adaptive Page Replacement”, Proc. the 1999 ACM SIGMETRICS Conference on Mesurement and Modeling of Computer Systems, pp. 122-133(1999)