プロセスとファイルキャッシュを共有するオンメモリファイル機能の提案

枡田圭祐[†]　谷口秀夫[†]

マイクロカーネル構造の OS では，OS サーバ間の通信が頻発するため，OS サーバ間の高速な通信機構が提案されている．これに対し，AP プロセスがファイル操作を行なう際，ファイルキャッシュを有する OS サーバはファイルキャッシュのデータを AP プロセスのメモリ空間に複写する必要があり，性能が低下する．しかし，有効な対処法は提案されていない．そこで，本論文では，ファイル操作時の実メモリ間複写回数を削減するオンメモリファイル機能を提案する．この機能は，OS サーバのファイルキャッシュをプロセスと共有させることで，データ読み書き時の実メモリ間複写回数を削減する．また，オンメモリファイル機能を ***AnT*** に実現し，MINIX（3.2.0），および FreeBSD（6.3-RELEASE）のファイル操作機能と比較し，その有効性を示す．

Proposal of On Memory File Mechanism that Shares File Cache with Processes

Keisuke MASUDA[†] and Hideo TANIGUCHI[†]

In OS based on microkernel architecture, since intercommunication between OS servers occurs frequently, the high-speed mechanism has been proposed. On the other hand, OS server with file cache is necessary to copy data to memory space of AP process, and performance deteriorates. However, effective actions are not proposed. Therefore, we propose the on memory file mechanism to reduce data copy between memory spaces. This mechanism shares file cache with processes, and reduces data copy between memory spaces in reading and writing. We realize the on memory file mechanism in ***AnT***, and compare with file operation mechanism of MINIX (3.2.0) and FreeBSD (6.3-RELEASE) to show efficiency.

1. はじめに

高い適応性と堅牢性を実現するオペレーティングシステム（以降，OS）のプログラム構造として，マイクロカーネル構造[1),2)] がある．マイクロカーネル構造は，例外処理や割込処理といった最小限の OS 機能をカーネルとして実現し，ファイル管理やディスクドライバといった大半の OS 機能をプロセス（以降，OS サーバ）として実現するプログラム構造である．MINIX[3),4)]，Mach[5)]，L4Linux[6),7)]，および ***AnT***[8)] では，マイクロカーネル構造を採用し，OS の各種機能をプロセスとして実現している．

マイクロカーネル構造の OS では，OS サーバ間の通信が頻発するため，FreeBSD のようなモノリシックカーネル構造の OS に比べ，データの実メモリ間複写が多発し，性能が低下する．そこで，この性能低下を抑制する機構として，OS サーバ間の高速な通信機構[8)] が提案されている．これに対し，応用プログラム（以降，AP）のプロセスがファイル操作を行なう場合，ファイルキャッシュ機能を有する OS サーバは，ファイルキャッシュのデータを AP プロセスのメモリ空間に複写する必要があり，性能が低下する．

従来から，OS 処理の主なオーバヘッドは，データの複写といわれている．これは，データの実メモリ間複写，および実メモリと外部記憶装置間の入出力に分類できる．後者のオーバヘッドを削減できる代表的な機能として，ファイル管理のファイルキャッシュ機能がある．しかし，ファイルキャッシュ機能は，入出力回数を削減できるものの，実メモリ間複写回数を削減できない．一方，ファイル参照時の実メモリ間複写回数を削減できる機能として，メモリマップドファイル（MMF）機能[9)] がある．しかし，MMF 機能は，ファイル参照に特化した機能であり，ファイルのデータ更新などを行なえない欠点がある．なお，外部記憶装置との実入出力法を工夫する研究は多数ある[10)~13)] が，

† 岡山大学大学院自然科学研究科
Graduate School of Natural Science and Technology, Okayama University

表 1　ファイル操作機能の比較
Table 1　Comparison of file operation mechanism.

	実メモリ間複写	ファイルキャッシュ	i ノードの更新	読み書きのデータ単位
通常入出力機能	2 回	有効	可能	1Byte
直接入出力機能	0 回	無効	可能	ブロック
MMF 機能	0 回	有効	不可能	ページ
OMF 機能	0 回	有効	可能	ブロック（ただし，ライブラリでは 1Byte）

いずれも，実メモリ間複写回数を削減するものではない．

そこで，本論文では，ファイル操作時の実メモリ間複写回数を削減するオンメモリファイル（OMF）機能を提案する．この機能は，OS サーバのファイルキャッシュをプロセスと共有させることで，データ読み書き時の実メモリ間複写回数を削減する．また，オンメモリファイル機能を ***AnT*** に実現し，マイクロカーネル構造を有する MINIX（3.2.0），およびモノリシックカーネル構造を有する FreeBSD（6.3-RELEASE）のファイル操作機能と比較する．

2. 関 連 機 能

既存のファイル操作機能として，通常入出力機能，直接入出力機能，およびメモリマップドファイル（MMF）機能がある．

通常入出力機能は，ファイル管理のファイルキャッシュ機能を利用している．このため，AP プロセスのファイルデータの参照は，ファイルキャッシュから AP プロセスのメモリ空間へデータを複写し，参照する．更新は，メモリ空間上のデータを更新し，メモリ空間からファイルキャッシュにデータを複写する．提供インタフェースとして，read()，write()，fread()，fwrite() がある．

直接入出力機能は，ファイル管理のファイルキャッシュ機能を利用せず，外部記憶装置と AP プロセスのメモリ空間の間で直接データを授受する機能である．例えば，この機能は，データベース管理システム（DBMS）のように，プログラム内にファイルキャッシュ機能を有し，実入出力回数を削減する機構を持つプログラムで利用されている．

MMF 機能は，共有メモリを利用してファイルキャッシュ機能を実現しており，ファイルデータを外部記憶装置から共有メモリ上に読み出し，AP プロセスが参照できるようにする機能である．例えば，AP プロセスの手続き部（テキスト部）として利用されている．

各機能の比較を**表 1** に示す．通常入出力機能は，1Byte 単位でファイルデータを読み書きできる利点を持つ．しかし，ファイルデータを読み書きする際に実メモリ間複写が必ず発生する欠点がある．直接入出力機能は，ファイルデータを読み書きする際に実メモリ間複写が発生しない．しかし，入出力するデータのサイズがブロック単位に制限される欠点がある．MMF 機能は，ファイルデータを読み書きする際に実メモリ間複写が発生しない．しかし，ファイルの情報を管理する i ノードの更新を行なわないため，ファイルサイズの拡縮ができない欠点がある．また，読み書きするデータのサイズがページ単位に制限される欠点もある．

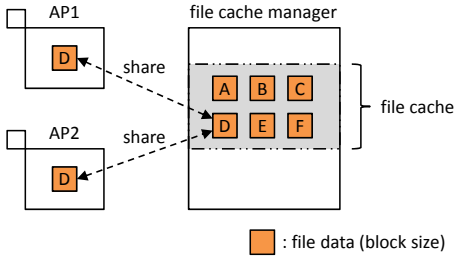


図 1　オンメモリファイル機能
Fig. 1　On memory file mechanism.

したがって，MMF 機能の欠点を克服した新たな機能を実現できれば，独自のキャッシュ機構を持たない多くのサービス処理において，実メモリ間複写が発生しない形で，高速にファイルの参照や更新を行なうことができる．

3. オンメモリファイル機能

3.1 基 本 機 構

オンメモリファイル（OMF）機能は，以下の考え方に基づいている．

(1)　プロセスとファイルキャッシュを共有する．
(2)　ファイルサイズを拡縮できる．
(3)　ファイルの参照日時や更新日時を更新できる．

OMF 機能の基本方式を**図 1** に示す．OMF 機能は，プロセスとファイルキャッシュを共有する．つまり，AP プロセスのデータの参照や更新は，ファイルデータを格納する実メモリ空間のアドレスを AP プロセスの仮想メモリ空間へマッピングして行なう．したがって，通常入出力機能と異なり，ファイルキャッシュと AP プロセスの仮想メモリ空間の間でデータの複写は

表 2 システムコールのインタフェース
Table 2 System call interface.

機能	形式
データの参照 （ブロック単位）	readblock(fd, size, blkoff); fd: ファイル識別子 size: データのサイズ（ブロック単位） blkoff: 参照開始位置（ブロック単位）
外部記憶装置への データの書き出し	syncblock(fd, *buf, size, blkoff, flg); fd: ファイル識別子 *buf: データを格納するメモリ空間 size: データのサイズ（バイト単位） blkoff: 更新開始位置（ブロック単位） flg: ファイルサイズを拡縮するか否か

発生しない．なお，OMF 機能は，ファイルデータをブロック（4KBytes）単位で管理する．

また，OMF 機能は，ファイルサイズを拡縮できる．具体的には，ファイルサイズを拡縮するために，i ノード内のファイルサイズやファイルデータの外部記憶装置上の格納位置を更新する．さらに，OMF 機能は，ファイルの参照日時や更新日時を更新できる．参照日時や更新日時を更新するために，i ノード内の参照日時や更新日時を更新する．これらの i ノードの更新は，ファイルデータを外部記憶装置へ書き出す際に行なう．なお，外部記憶装置への書き出しを行なわない場合，参照日時や更新日時は更新されない．

3.2 提供インタフェース

OMF 機能が提供するシステムコールのインタフェースを**表 2** に示す．readblock() は，ファイルキャッシュ内のファイルデータを AP プロセスの仮想メモリ空間にブロック単位でマッピングする．なお，ファイルキャッシュにファイルデータが存在しない場合，外部記憶装置からファイルキャッシュへファイルデータを読み込む．syncblock() は，ファイルキャッシュ内のファイルデータを外部記憶装置へ書き出す．

当然のことながら，OS サーバのファイルキャッシュをプロセスと共有するため，プロセスがファイルデータを更新すると，ファイルキャッシュの内容も更新される．したがって，データをファイルキャッシュへ複写するインタフェース（通常入出力機能の write() に相当）は不要である．

システムコールの入出力単位は，ブロック単位である．このため，バイト単位でファイルデータを参照する機能をライブラリとして実現する．このインタフェースを readbyte() と名付け，**表 3** に示す．readbyte() は，ファイルのブロックデータを初めて参照する際に readblock() を発行し，ブロックデータをライブラリ内にバッファリングする．これにより，以降でデータを参照する際に，カーネルの呼び出しを削減できる．

表 3 ライブラリコールのインタフェース
Table 3 Library call interface.

機能	形式
データの参照 （バイト単位）	readbyte(fd, size, offset); fd: ファイル識別子 size: データのサイズ（バイト単位） offset: 参照開始位置（バイト単位）

3.3 処理流れの比較

最初に，バイト単位のデータ操作について議論する．

readbyte() と通常入出力機能インタフェース（read(), fread()）について，AP プロセスがデータを参照する場合の比較を**表 4** に示す．read() を用いる場合，データ格納域をメモリ空間に確保し，read() システムコールを発行する．ここで，データ格納域は，既にデータ部またはスタック部に存在すると仮定する（以降の場合においても，これを仮定する）．なお，read() 処理による実メモリ間複写が発生する．fread() を用いる場合，データ格納域をメモリ空間に確保し，fread() を発行する．なお，fread() 処理において，実メモリ間複写が発生し，さらに，データ格納のための領域確保システムコール，および read() システムコールを発行（カーネル呼び出し）する場合がある．readbyte() を用いる場合，ライブラリ内のデータ格納位置を通知するものであるため，データ格納域の確保は不要であり，実メモリ間複写も発生しない．ただし，ライブラリ内に当該データが存在しない場合は，readblock() システムコールを発行する．

また，readbyte() と通常入出力機能インタフェース（read()，write()，fread()，fwrite()）について，AP プロセスがデータを読み書きする場合の比較を**表 5** に示す．read() / write() を用いる場合，データ格納域をメモリ空間に確保し，read() システムコールを発行し，メモリ空間内のデータを書き替え，write() システムコールを発行する．なお，メモリ空間内のデータの書き替えは，AP プロセスの有するデータのメモリ空間内への複写であるといえる．このため，read() 処理，書き替え処理，および write() 処理による実メモリ間複写が発生する．fread() / fwrite() を用いる場合，データ格納域をメモリ空間に確保し，fread() を発行し，メモリ空間内のデータを書き替え，fwrite() を発行する．なお，fread() 処理，書き替え処理，および fwrite() 処理において実メモリ間複写が発生し，さらに，データ格納のための領域確保システムコール，read() システムコール，および write() システムコールを発行（カーネル呼び出し）する場合がある．readbyte() を用いる場合，ライブラリ内のデータ格納

表 4 データ参照の比較
Table 4 Comparison of data read.

	インタフェース	データ格納域確保	実メモリ間複写	カーネル呼び出し
通常入出力機能	read()	0 回	1 回	1 回
	fread()	0〜1 回	1〜2 回	0〜2 回
OMF 機能	readbyte()	0 回	0 回	0〜1 回

表 5 データ読み書きの比較
Table 5 Comparison of data r/w.

	インタフェース	データ格納域確保	実メモリ間複写	カーネル呼び出し
通常入出力機能	read() / write()	0 回	3 回	2 回
	fread() / fwrite()	0〜1 回	3〜5 回	0〜3 回
OMF 機能	readbyte()	0 回	1 回	0〜1 回

表 6 新規データによるデータ更新の比較
Table 6 Comparison of data update by new data.

	インタフェース	実メモリ間複写	カーネル呼び出し
通常入出力機能	write()	1 回	1 回
	fwrite()	1〜2 回	0〜1 回
OMF 機能	syncblock()	0〜1 回	1 回

位置を通知するものであるため，データ格納域の確保は不要であり，readbyte() 発行時に実メモリ間複写は発生しない．ただし，ライブラリ内に当該データが存在しない場合は，readblock() システムコールを発行する．なお，書き替え処理により，データ格納域への実メモリ間複写が発生する．また，OS サーバのファイルキャッシュをプロセスと共有するため，データをファイルキャッシュへ複写する処理（write() システムコールに相当）は不要である．

以上より，いずれの場合も実メモリ間複写回数やカーネルの呼び出し回数は，OMF 機能を用いた方式が通常入出力機能を用いた方式より少ないことがわかる．

次に，新規データによるファイルデータ更新の処理について考察する．syncblock() と通常入出力機能のインタフェース（write()，fwrite()）について，AP プロセスが新規データを用いてファイルデータを更新する場合の比較を**表 6** に示す．write() を用いる場合，write() システムコールを発行する．なお，write() 処理による実メモリ間複写が発生する．fwrite() を用いる場合，fwrite() を発行する．なお，fwrite() 処理において実メモリ間複写が発生し，さらに，write() システムコールを発行（カーネル呼び出し）する場合がある．syncblock() を用いる場合，syncblock() システムコールを発行する．なお，更新対象のファイルデータがファイルキャッシュ上に存在する場合，新規データのファイルキャッシュへの複写が発生する．つまり，実メモリ間複写が発生する．

以上より，新規データによりファイルデータを更新する場合，OMF 機能を用いた方式は，通常入出力機能を用いた方式と同様に，実メモリ間複写が発生する．また，fwrite() を用いた方式と異なり，毎回システムコールを発行する．ただし，システムコールを発行するのは外部記憶装置への書き出し時のみである．

3.4 期待される効果

表 1 に示したように，OMF 機能は，既存のファイル操作機能に比べ，以下の特徴を持つ．

(1) 通常入出力機能に比べ，ファイルデータを読み書きする際に実メモリ間複写が発生しない．

(2) 直接入出力機能に比べ，ファイルキャッシュを利用するため，独自のキャッシュ機構を持たないプロセスにとって有効である．

(3) MMF 機能に比べ，i ノードの更新を行なうため，ファイルサイズを拡縮でき，ファイルの参照だけでなく更新にも有効である．

また，OMF 機能が有効な処理の例を以下に示す．

(例 1) ファイルデータの一部だけをランダムアクセスする処理の場合，ファイルデータをメモリの読み書きと同様に操作できる．

(例 2) 複数のプロセスが同一ファイルの同一ブロック内の別領域を操作する処理の場合，プロセスとファイルキャッシュを共有しており，プロセスが更新した内容は他のプロセスでも即座に反映されるため，排他制御を考慮する必要がない．当然のことながら，同一領域に同時に書き込む場合は，排他制御が必要になる．

(例 3) 複数のプロセスが大きなサイズの同一ファイル

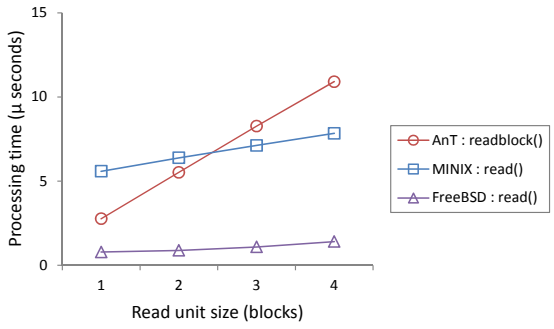


図 2 n ブロックのデータ参照処理時間
Fig. 2 Processing time of n blocks read.

を操作する処理の場合，プロセスとファイルキャッシュを共有するため，メモリの利用効率が良い．

4. 評　　価

4.1 観点と評価環境

表 1 に基づき，ファイルキャッシュが有効で i ノードの更新が可能な OMF 機能と通常入出力機能について性能を比較する．実メモリ間複写回数に着目するため，ファイルキャッシュのヒット率向上のための機構，および実入出力処理機構による影響を排除する．具体的には，すべてのデータがファイルキャッシュ上に存在する状況で性能を測定し比較する．

OMF 機能を ***AnT*** に実現し，MINIX（3.2.0），および FreeBSD（6.3-RELEASE）の通常入出力機能と比較する．各 OS を Core i7-2600（3.4GHz）の計算機で走行させた．

まず，単一の AP プロセスが単一のファイルデータを操作する基本的な性能を明らかにするため，参照性能，更新性能，ランダムなデータ参照，および同一ブロック内の別領域の読み書きについて評価する．次に，メモリの利用効率について評価する．

なお，OMF 機能と MMF 機能を用いて 1 ブロックのデータ参照処理時間を測定したところ，***AnT***（OMF 機能）では 2.76μ 秒，FreeBSD（MMF 機能）では 0.87μ 秒であった．***AnT*** と FreeBSD の差（1.89μ 秒）は，マイクロカーネル構造とモノリシックカーネル構造の違いによるものである．なお，MINIX は MMF 機能を有しない．

4.2 基 本 性 能

4.2.1 参 照 性 能

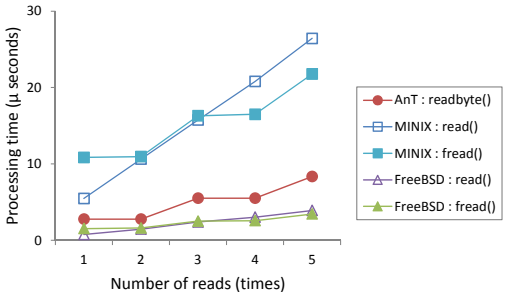
最初に，プロセスが一度に n ブロック（1 ブロックは 4KBytes）のデータを参照する処理時間を**図 2** に示す．つまり，readblock() と read() の比較である．図 2 より，以下のことがわかる．

(1)　参照単位が小さい場合，***AnT***（OMF 機能）は MINIX（通常入出力機能）より高性能である．1 ブロックのデータの参照処理時間は，***AnT*** で 2.76μ 秒，MINIX で 5.58μ 秒，FreeBSD で 0.78μ 秒である．***AnT*** の処理時間が MINIX より短い理由は，サーバ間通信が高速であり，かつ実メモリ間複写が発生しないためである．また，***AnT*** と MINIX の処理時間が FreeBSD より長い理由は，***AnT*** と MINIX がマイクロカーネル構造を有するため，データ参照時のサーバ間通信により処理時間が長くなるのに対し，FreeBSD はモノリシックカーネル構造を有するため，データ参照時にサーバ間通信が発生しないことに起因する．

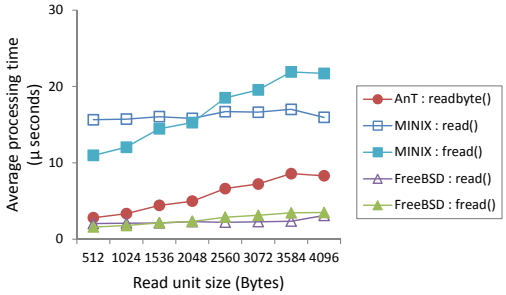
(2)　***AnT***（OMF 機能）の処理時間は，参照単位の増加に伴い，大きく増加する．一方，MINIX（通常入出力機能）と FreeBSD（通常入出力機能）の処理時間の増加は少ない．これは，次の理由による．***AnT*** のサーバプログラム間通信機構[8] では，実アドレス上で連続した n ブロックのデータ格納域を 1 つしか渡すことができない．一方，ファイルのデータ格納域（ファイルキャッシュ）は，1 ブロックごとに，実アドレス上で非連続な領域として管理している．このため，1 回のサーバ間通信で授受できるデータ格納域は，1 ブロックのみに制限される．したがって，n ブロックのデータを参照する場合，1 ブロックの場合に比べ，n 倍のサーバ間通信が必要になってしまう．つまり，***AnT*** のデータ参照処理時間は，参照単位が 1 ブロック増加するにつれて，1 ブロックのデータ参照処理時間（2.76μ 秒）ずつ増加する．したがって，実アドレス上で非連続（ただし，仮想アドレス上で連続）な n ブロックのデータ格納域をプロセス間通信時に渡すことのできる OS（MINIX や FreeBSD）に OMF 機能を実現した場合，n ブロックのデータを参照する際に，1 ブロックの場合と同じ回数のサーバ間通信で参照でき，処理時間は一定になる．なお，MINIX と FreeBSD の処理時間の増加は，実メモリ間複写時間の増加に起因する．具体的には，参照単位が 1 ブロック増加するのに伴い，MINIX では 0.75μ 秒，FreeBSD では 0.20μ 秒増加する．

次に，プロセスがデータを nBytes 単位で参照する処理時間を**図 3** に示す．図 3(a) は 2048Bytes 単位でシーケンシャルに参照を繰り返した場合の各処理時間，図 3(b) は (a) の処理時間の平均値（5 回分）を各参照単位について示したものである．

(1)　参照回数や参照単位に関係なく，***AnT***（OMF 機能）は MINIX（通常入出力機能）より高性能である．これは，システムコール発行処理時間が ***AnT***（2.76μ

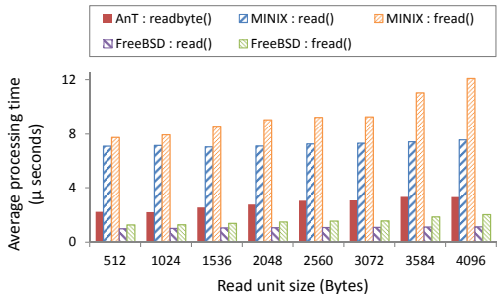


(a) 2048 Bytes unit read

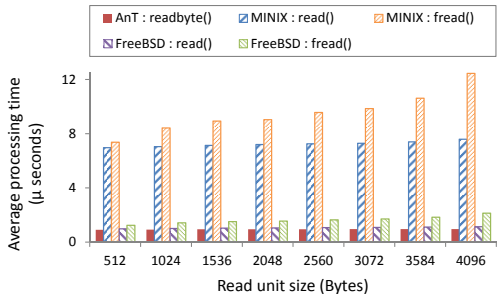


(b) Average of 5 times

図 3 nBytes 単位でのデータ参照処理時間
Fig. 3 Processing time of n Bytes unit read.



(a) Average of 10 times



(b) Average of 100 times

図 4 nBytes 単位でのランダムなデータ参照処理時間
Fig. 4 Processing time of random read of n Bytes unit.

秒）は MINIX（4.85μ 秒）より短く，MINIX の場合は実メモリ間複写が発生するためである．

（2） readbyte()，fread() を用いた場合，数回に 1 回システムコールを発行する．このため，処理時間は，数回に 1 回大きく増加する．具体的には，readbyte() の場合はシステムコール処理時間，fread() の場合はシステムコール処理時間と実メモリ間複写時間だけ増加する．一方，read() を用いた場合，毎回システムコールを発行する．このため，処理時間は参照回数に比例して増加する．

4.2.2 更 新 性 能

プロセスが 1 ブロック，2 ブロック，および 4 ブロックのデータを更新する処理時間について議論する．実測の結果，MINIX（通常入出力機能）の場合，それぞれ 5.38μ 秒，6.37μ 秒，および 8.09μ 秒であり，FreeBSD（通常入出力機能）の場合，それぞれ 1.00μ 秒，1.15μ 秒，および 1.49μ 秒であった．これに対し，***AnT***（OMF 機能）の場合，プロセスとファイルキャッシュを共有するため，プロセスがデータをファイルキャッシュへ複写する処理（write() に相当）は不要である．したがって，***AnT*** の更新処理時間は 0 と考えられ，MINIX と FreeBSD に比べ，更新性能は高い．

4.2.3 ランダムなデータ参照

128KBytes のファイルにおいて，ランダムな位置から nBytes ずつ参照する処理時間を**図 4** に示す．図 4(a) は参照回数が 10 回，図 4(b) は参照回数が 100 回の場合の平均処理時間である．なお，参照したブロックデータ数は，図 4(a) では 7 個，図 4(b) では 30 個であった．

（1） 参照回数によらず，***AnT***（OMF 機能）は MINIX（通常入出力機能）より高性能である．これは，***AnT*** のサーバ間通信が高速であり，かつ実メモリ間複写が発生しないことに起因する．なお，OMF 機能（***AnT***）では readbyte() の引数として参照開始位置を指定できるが，通常入出力機能（MINIX，FreeBSD）では lseek() システムコールや fseek() により参照開始位置を設定する必要がある．これによる処理時間の増加も含まれる．

（2） 参照回数が多い場合，***AnT***（OMF 機能）は FreeBSD（通常入出力機能）より高性能である．これは，OMF 機能の方が少ないメモリ量で多くのブロックデータをバッファリングできるためである．fread() と readbyte() において，1 ブロックのデータをバッファリングするために必要なメモリ量を考察する．fread() では，ブロックデータを特定するための情報（8Bytes）とブロックデータ（4KBytes）を格納するための量

(8Bytes+4KBytes) が必要になる．一方，readbyte() では，ブロックデータを特定するための情報（8Bytes）とブロックデータを指すポインタ（4Bytes）を格納する量（12Bytes=8Bytes+4Bytes）だけで良い．例えば，8Bytes+4KBytes の量でバッファリングできるブロック数は，fread() では 1 ブロックになるのに対し，readbyte() では 342（= (8Bytes+4KBytes)/12Bytes）ブロックになる．当然のことながら，参照回数が増加すると，参照するブロックデータ数も増加する．この際，ライブラリ内にバッファリングできるブロックデータ数が多いほど，システムコール発行回数が少なく，処理時間が短くなる．したがって，参照回数が多い場合，少ないメモリ量で多くのブロックデータをバッファリングできる ***AnT*** の方が高性能になる．

(3) 参照回数が多い場合，参照単位によらず，***AnT***（OMF 機能）の処理時間は一定である．***AnT*** では，ライブラリ内のバッファのサイズを 256Bytes に固定している．このため，参照回数によらずメモリ量は一定である．また，256Bytes の領域にバッファリングできるブロックデータ数は 32（=256Bytes/12Bytes）である．したがって，100 回参照した場合（ブロックデータ数は 30）でも，すべてのブロックデータをバッファリングでき，参照単位によらず処理時間は一定になる．これに対し，fread() では，初回発行時の引数のサイズをもとに，ライブラリ内にバッファが確保される．このため，***AnT*** と同様に，参照回数によらずメモリ量は一定である．ただし，参照回数の増加に伴い参照するブロックデータ数が増加するため，ライブラリ内のバッファが何度も上書き更新され，システムコール発行回数を削減できず，処理時間が増加する．

4.2.4 同一ブロック内の別領域の読み書き

複数の AP プロセスが同一ファイルの同一ブロック内の別領域を読み書きする処理では，各 AP プロセスが独立に読み込んだメモリ上のデータを読み書きする．通常入出力機能（MINIX，FreeBSD）と OMF 機能（***AnT***）について，処理の大きな違いを以下に示す．

(1) 通常入出力機能（MINIX，FreeBSD）では，AP プロセスのメモリ空間からファイルキャッシュへデータを複写し，データを更新する．これに対し，OMF 機能（***AnT***）では，プロセスとファイルキャッシュを共有するため，実メモリ間複写は発生しない．

(2) 通常入出力機能（MINIX，FreeBSD）では，データの参照開始位置や更新開始位置を設定（lseek() や fseek() を発行）する．これに対し，OMF 機能（***AnT***）では，readbyte() の引数として，参照開始位置を指定する．また，プロセスとファイルキャッシュを共有するため，更新開始位置の設定は不要である．

(3) 通常入出力機能（MINIX，FreeBSD）では，書き替える領域が別領域であっても，同一ブロック内であれば，データを読み書きする間，排他制御（P 操作と V 操作）が必要になる．これに対し，OMF 機能（***AnT***）では，プロセスとファイルキャッシュを共有するため，別領域を書き替える場合，排他制御は不要である．当然のことながら，同一領域を書き替える場合は，排他制御が必要になる．

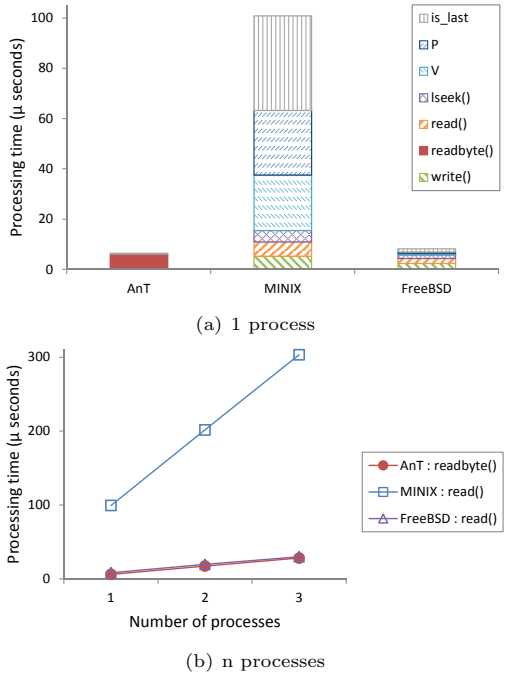


図 5 同一ブロック内の別領域の読み書き処理時間
Fig. 5 Processing time of r/w other area in same block.

複数の AP プロセスが同一ブロック内の別々の 1Byte 領域を参照し，書き替え，更新する処理時間を**図 5** に示す．図 5(a) は AP プロセス数が 1 の場合，図 5(b) は AP プロセス数が複数（1〜3）の場合である．なお，AP プロセスのメモリ上のデータ書き替え時間は非常に短いため，以降の記述で省略している．図 5 より，以下のことがわかる．

(1) 図 5(a) より，***AnT***（OMF 機能）は最も高性能である．各 OS の処理時間は，***AnT*** で 6.31μ 秒，MINIX で 99.47μ 秒，FreeBSD で 8.42μ 秒であった．***AnT*** の処理時間が MINIX より短い理由は，実メモリ間複写がなく，カーネル呼び出し回数が少なく，排他制御が不要であり，かつサーバ間通信が高速であることによる．また，***AnT*** の処理時間が FreeBSD より短い理由は，実メモリ間複写がなく，カーネル呼び

出し回数が少なく，かつ排他制御が不要なことによる．(2) 図 5(b) より，各 OS の n プロセスの場合の処理時間は，1 プロセスの場合の n 倍である．したがって，プロセス数が増加しても，***AnT***（OMF 機能）は最も高性能である．

4.3 メモリの利用効率

通常入出力機能の場合，メモリ量は，ファイルキャッシュとして利用する量に加え，プロセスがデータを格納するための量が，プロセスと同じ数だけ必要になる．これに対し，OMF 機能の場合，プロセスとファイルキャッシュを共有するため，メモリ量は，ファイルキャッシュとして利用する量のみで良い．また，先に説明したように，ライブラリ内のバッファは，通常入出力機能より少ないメモリ量で多くのブロックデータをバッファリングできる．したがって，メモリの利用効率は，OMF 機能を用いた方式が通常入出力機能を用いた方式より良いことがわかる．

5. おわりに

プロセスとファイルキャッシュを共有するオンメモリファイル（OMF）機能を提案した．OMF 機能は，ファイルのデータ読み書き時に，実メモリ間複写回数を削減できる．特に，プロセスはデータをファイルキャッシュへ複写する処理（write() システムコール相当）が不要である．さらに，ファイルサイズを拡縮でき，参照日時や更新日時を更新できる．

マイクロカーネル構造を有する ***AnT*** に OMF 機能を実現し，MINIX，および FreeBSD の通常入出力機能と比較評価した．

基本性能の評価において，参照単位が 1 ブロックの場合，***AnT***（OMF 機能）の処理時間は 2.76μ 秒であり，MINIX（5.58μ 秒）より短かった．なお，参照単位の増加に伴い，***AnT***（OMF 機能）の処理時間が大きく増加した原因は，***AnT*** のサーバプログラム間通信機構[8] が 1 ブロック単位の通信を基本としているためである．ランダムなデータ参照では，参照回数によらず，***AnT***（OMF 機能）の処理時間は一定であった．一方，MINIX と FreeBSD の処理時間は参照回数の増加に伴い増加した．さらに，複数プロセスによる同一ブロック内の別領域の読み書きでは，プロセス数によらず，***AnT***（OMF 機能）の処理時間が最も短かった．いずれも，実メモリ間複写を削減した効果である．次に，メモリの利用効率の評価において，OMF 機能を用いた方式が，通常入出力機能を用いた方式より効率が良いことを明らかにした．

謝辞 本研究の一部は，科学研究費補助金基盤研究 (B)（課題番号：24300008）による．

参考文献

1) Liedtke, J.: Toward real microkernels, Commun. ACM, Vol.39, No.9, pp.70-77 (1996).
2) Tanenbaum, A.S., Herder, J.N. and Bos, H.: Can we make operating systems reliable and secure?, IEEE Computer Magazine, Vol.39, No.5, pp.44-51 (2006).
3) Tanenbaum, A.S., Herder, J.N., Bos, H., Gras, B. and Homburg, P.: Modular system programming in MINIX 3, The USENIX Magazine, Vol.31, No.2, pp.19-28 (2006).
4) Tanenbaum, A.S., Herder, J.N., Bos, H., Gras, B. and Homburg, P.: Roadmap to a failure-resilient operating system, The USENIX Magazine, Vol.32, No.1, pp.14-20 (2007).
5) Black, D.L., Golub, D.B., Julin, D.P., Rashid, R.F., Draves, R.P., Dean, R.W., Forin, A., Barrera, J., Tokuda, H., Malan, G.R. and Bohman, D.: Microkernel operating system architecture and mach, J. Inf. Process., Vol.14, No.4, pp.442-453 (1992).
6) Liedtke, J.: Improving IPC by kernel design, Proc. 14th ACM Symp. Operating System Principles, ACM Press, pp.175-188 (1994).
7) Hartig, H., Hohmuth, M., Liedtke, J., Wolter, J. and Schonberg, S.: The performance of microkernel-based systems, Proc. 16th ACM Symp. Operating System Principles, pp.66-77 (1997).
8) 岡本幸大, 谷口秀夫: ***AnT*** オペレーティングシステムにおける高速なサーバプログラム間通信機構の実現と評価, 電子情報通信学会論文誌 (D), Vol.J93-D, No.10, pp.1977-1989 (2010).
9) Gallmeister, B.O.: POSIX.4, O'Reilly, pp.128-129 and 389-391 (1995).
10) Mckusick, M.K., Karels, M.J., Bostic, K.: A Pageable Memory Based Filesystem, Proceedings of the Summer 1990 USENIX Technical Conference, pp.137-144 (1990).
11) Pai, V.S., Druschel, P., Zwaenepoel, W.: IO-Lite: A Unified I/O Buffering and Caching System, ACM Transactions on Computer Systems, Vol.18, No.1, pp.37-66 (1999).
12) Thomasian, A.: Survey and Analysis of Disk Scheduling Methods, ACM SIGARCH Computer Architecture News, Vol.39, No.2, pp.8-25 (2011).
13) Kang, Y., Yang, J., Miller, E.L.: Object-based SCM: An Efficient Interface for Storage Class Memories, 27th IEEE Conference on Mass Storage Systems and Technologies (2011).