PostgreSQL 解析資料

～ ストレージマネージャ(1) ～

(株) NTT データ
ビジネス開発事業本部 システム方式 BU
井久保 寛明

1. はじめに

本資料では、PostgreSQL の 狭い意味での ストレージマネージャについて紹介する。ストレージマネージャというと、インデックスのデータ構造からファイルの格納、I/O の実装まで含めて広く言う場合もある。しかし本資料が対象とするのは、storage/smgr を中心とした、OS に対して I/O を起こすような低レベルの部分だけである。他の部分に関しては、追って他のドキュメントにまとめる予定である。

1.1. 対象モジュール

対象としているのは、具体的には storage/smgr と storage/file 以下のモジュールである。

2. storage/smgr

storage/smgr 配下のモジュールでは、抽象的なファイルI/Oのインタフェースを定義することで、HDD, メモリストレージ[1]などの実ストレージを、他のレイヤーから隠蔽している。また、このレイヤーで、削除されたリレーションに対応するファイルを消すタイミングの管理も行っている。

図 1 の一番上の層に書いてある、smgropen(), smgrread() などのインタフェースが、storage/smgr の提供している外部インタフェースになる。モジュール内部でも、関数呼び出しの部分を関数ポインタの構造体 f_smgr として定義し、その配列 smgrsw[] を用いることで、仮想的な関数インタフェースを定義している。これが、図１の２層目にある smgr_open, smgr_read などである。これら関数呼び出しの仮想インタフェースの実体は、md.c や mm.c で定義される関数である。
storage/file/fd.c の提供する仮想ファイルデスクリプタについては、３章に書いてあるので、そちらを参照して欲しい。

図１で誤解しないで頂きたいのは、xxxopen() のような関数が、必ずしもすぐに次のレイヤーの同じような名称の関数を呼ぶわけではないというところである。具体的な例を挙げると、smgrunlink() は呼び出されると削除リストにリレーションを登録するだけで、実際の mdunlink() を呼び出すのは、トランザクション終了時に smgrDoPendingDeletes()が呼び出されたときである。このあたりは、2.2 削除リレーションリストに詳しく書いてある。

[1] PostgreSQLでは、メインメモリをストレージとして使う実装も入っている。

Copyright © 2003 NTTDATA

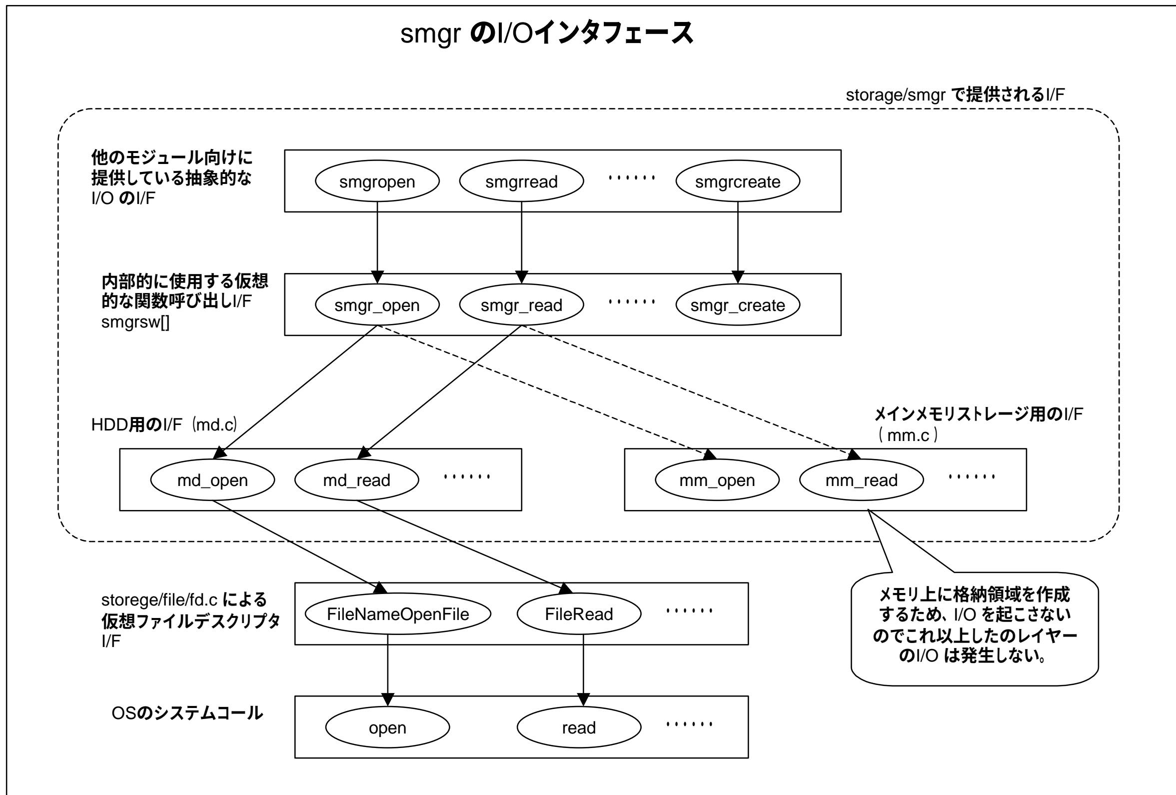

図 1

storage/smgr 配下ファイルは次のようなものである。

ファイル	
smgr.c	ストレージマネージャのファイル I/O インタフェース
smgrtype.c	複数種類のストレージマネージャを管理するための補助モジュール
md.c	HDD 上に格納されているリレーションの I/O の実装
mm.c	メインメモリ上にリレーションを格納するための I/O 部分の実装

PostgreSQL では、標準で "magnetic disk" と "main memory" の２つのメディアに対応したストレージモジュールが用意してあるが、STABLE_MEMORY_STORAGE の設定をしてコンパイルしないと、"main memory" の方は使用できない。つまり、現状の PostgreSQL は、通常 HDD しか使用できない。また、mm.c を調べたところ、標準では 2 リレーション、10 ブロックのメモリしか管理できないようになっているので、実際にメモリをストレージとして利用するためには、ソースコードに手を入れる必要がある。

2.1. 仮想ファイル I/O インタフェース (smgr.c)

smgr.c は storage/smgr のインタフェースを提供しているモジュールである。この中で、新しいリレーションが作成されてロールバックされた場合と、リレーションが削除されてロールバックされた場合の処理も実装されている。smgr.c が提供しているインタフェースは、次のようなものである。

Copyright © 2003 NTTDATA

smgrinit	全ストレージマネージャの初期化
smgrshutdown	全ストレージマネージャの終了処理
smgrcreate	リレーションの保存領域の生成(※ md の場合ファイルの生成)
smgrunlink	リレーションの保存領域の削除を行う。実際は、リストにつないでおいて、コミット時に削除は行う。
smgrextend	ファイルの末尾にブロックの追加を行う
smgropen	リレーションの保存領域のオープンを行う
smgrclose	リレーションの保存領域のクローズを行う
smgrread	リレーションの保存領域から１ブロック分のデータを読み出す
smgrwrite	リレーションの保存領域に１ブロック分のデータを書き込む
smgrblindwrt	リレーションの保存領域に１ブロック分のデータを書き込む[2]
smgrnblocks	リレーションに割り当てられている領域のブロック数を計算する
smgrtruncate	リレーションの保存領域のサイズを、指定されたサイズに縮める
smgrDoPendingDeletes	トランザクション終了時に、削除リストに繋がっているファイルの削除を実行する。
smgrcommit	全ストレージマネージャのコミット処理を呼び出す(※ md では何もしない)
smgrabort	全ストレージマネージャのアボート処理を呼び出す(※ md では何もしない)
smgrsync	全ストレージマネージャの sync 処理を実行。チェックポイントの処理に使用。
smgr_redo	未実装。ストレージに対するリカバリを想定？
smgr_undo	未実装。ストレージに対するリカバリを想定？
smgr_desc	未実装。

PostgreSQL では、リレーションに割り当ててあるブロック数を、実ファイルのサイズから算出するため、ファイルの拡張、ファイルの末尾の切捨ては、重要な処理である。

ストレージマネージャを切り替えて呼び出すための仮想的なインタフェースは、次のような構造体で定義されている。

```
typedef struct f_smgr
{
    int         (*smgr_init) (void);            /* may be NULL */
    int         (*smgr_shutdown) (void);        /* may be NULL */
    int         (*smgr_create) (Relation reln);
    int         (*smgr_unlink) (RelFileNode rnode);
    int         (*smgr_extend) (Relation reln, BlockNumber blocknum, char *buffer);
    int         (*smgr_open) (Relation reln);
```

[2] smgrwrite との違いは、引数の指定方法。

Copyright © 2003 NTTDATA

```
    int         (*smgr_close) (Relation reln);
    int         (*smgr_read) (Relation reln, BlockNumber blocknum, char *buffer);
    int         (*smgr_write) (Relation reln, BlockNumber blocknum, char *buffer);
    int         (*smgr_flush) (Relation reln, BlockNumber blocknum, char *buffer);
    int         (*smgr_blindwrt) (RelFileNode rnode, BlockNumber blkno,
                                              char *buffer, bool dofsync);
    int         (*smgr_markdirty) (Relation reln, BlockNumber blkno);
    int         (*smgr_blindmarkdirty) (RelFileNode, BlockNumber blkno);
    BlockNumber (*smgr_nblocks) (Relation reln);
    BlockNumber (*smgr_truncate) (Relation reln, BlockNumber nblocks);
    int         (*smgr_commit) (void);  /* may be NULL */
    int         (*smgr_abort) (void);   /* may be NULL */
    int         (*smgr_sync) (void);
} f_smgr;
```

この仮想I/Oインタフェースから、実I/Oインタフェースへのマッピングは、次のように定義されている。

```
static f_smgr smgrsw[] = {
    /* magnetic disk */
    {mdinit, NULL, mdcreate, mdunlink, mdextend, mdopen, mdclose,
        mdread, mdwrite, mdflush, mdblindwrt, mdmarkdirty, mdblindmarkdirty,
        mdnblocks, mdtruncate, mdcommit, mdabort, mdsync
    },

#ifdef STABLE_MEMORY_STORAGE
    /* main memory */
    {mminit, mmshutdown, mmcreate, mmunlink, mmextend, mmopen, mmclose,
        mmread, mmwrite, mmflush, mmblindwrt, mmmarkdirty, mmblindmarkdirty,
    mmnblocks, NULL, mmcommit, mmabort},
#endif
};
```

これらは、(*(smgrsw[DEFAULT_SMGR].smgr_create)) (reln)) のように呼び出され、実体の mdcreate() や mmcreate()が呼び出される。DEFAULT_SMGR(= 0) でコンパイルされるので、デフォルトではHDD側の関数を使うようになる。HDDへのアクセスインタフェースの定義は、md.c で行われている。

2.2. 削除リレーションリスト

ストレージマネージャのもう１つの重要な役割は、削除リレーションリストの管理である。トランザクション中に作成したリレーションと削除したリレーションは、削除リストで管理する。
smgrcreate() でリレーションファイルが作成されたときは、すぐに物理ファイルが作成される。それと同時に、アボートされた場合に、削除できるように削除リストに登録しておく。逆に、smgrunlink() でリレーションファイルの削除が行われた場合、削除処理はすぐには行わず、削除リストに入れるだけである。そしてトランザクションがコミットされたら、その時点で物理ファイルを消すことができる。このように、実際の物理ファイルの削除処理は、トランザクション終了時の smgrDoPendingDeletes() のタイミングで行われる。

Copyright © 2003 NTTDATA

削除リレーションのリストは、次の構造体で管理される。

```
typedef struct PendingRelDelete
{
    RelFileNode relnode;            /* relation that may need to be deleted */
    int16       which;              /* which storage manager? */
    bool        isTemp;             /* is it a temporary relation? */
    bool        atCommit;           /* T=delete at commit; F=delete at abort */
    struct PendingRelDelete *next;  /* linked-list link */
} PendingRelDelete;

static PendingRelDelete *pendingDeletes = NULL; /* head of linked list */
```

次の図のように、コミット時に削除するものとアボート時に削除するものは同一のリストで管理され、フラグによって動作を分けている。

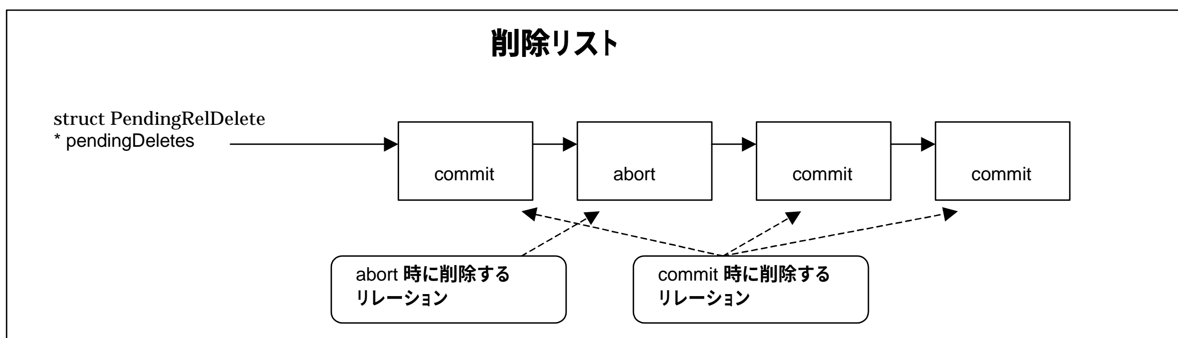

図 2

トランザクションをまたがって削除リストを維持する必要はないので、コミットの場合でも、アボートの場合でも、トランザクション終了時にこのリストは空になる。コミットの場合はコミットフラグのファイルだけ削除し、アボートの場合はアボートフラグのファイルだけ削除する。該当しないものは、読み捨てることになる。

2.3. 複数ストレージマネージャ管理の補助モジュール（smgrtype.c）

複数のストレージマネージャを扱うことができるようにするための補助モジュールである。ストレージマネージャ名からＩＤを取得したり、その逆を行ったりする関数と、ストレージマネージャの比較を行う簡単な関数が定義されているだけである。

新しいストレージを追加するには、smgr.c に定義を追加して、こちらに名前を登録することで使えるようになる。

2.4. md.c

磁気ディスクをストレージとして使うための I/O の実装である。storage/file/fd.c では、OS ファイルのデスクリプタを仮想化しているが、md.c では、これらを束ねて、リレーションごとの仮想ファイルデスクリプタとして管理する。リレーションの仮想ファイルデスクリプタとは、このモジュールの

Copyright © 2003 NTTDATA

Md_fdvec 配列へのエントリ番号である。
また、OS のファイルサイズの限界を超えるリレーションのファイルをサポートするため、リレーションファイルを 2GB 以下のセグメントに分けて保存する機能を提供する。データブロックのサイズ BLCKSZ と リレーションのセグメント数 RELSEG_SIZE の値は、include/pg_config.h に定義されている。storage/file/buffile.c では、RELSEG_SIZE * BLCKSZ = 1GB をファイルサイズの上限として扱っている。しかし、md.c では、LET_OS_MANAGE_FILESIZE が定義してあるかどうかで、動作が異なってくる。しかし、基本的に Linux では、LET_OS_MANAGE_FILESIZE は定義されていないため、管理は md.c で行うことになる。つまり、buffile.c と同様に、１セグメントは１ＧＢである。

ファイルデスクリプタの管理に使用される構造体と主なローカル変数は、次のようになっている。

```
typedef struct _MdfdVec
{
    int         mdfd_vfd;       /* fd number in vfd pool */
    int         mdfd_flags;     /* fd status flags */
    int         mdfd_nextFree;  /* link to next freelist member, if free */
#ifndef LET_OS_MANAGE_FILESIZE
    struct _MdfdVec *mdfd_chain;    /* for large relations */
#endif
} MdfdVec

static int  Nfds = 100;         /* 現在の Md_fdvec 配列の大きさ */
static MdfdVec *Md_fdvec = (MdfdVec *) NULL;
static int  Md_Free = -1;       /* フリーリストのエントリ位置 */
static int  CurFd = 0;          /* 最初の未使用 fdvec の index 番号 */
```

Md_fdvec による、仮想ファイルデスクリプタ全体の管理構造を表したのが次の図である。

Copyright © 2003 NTTDATA

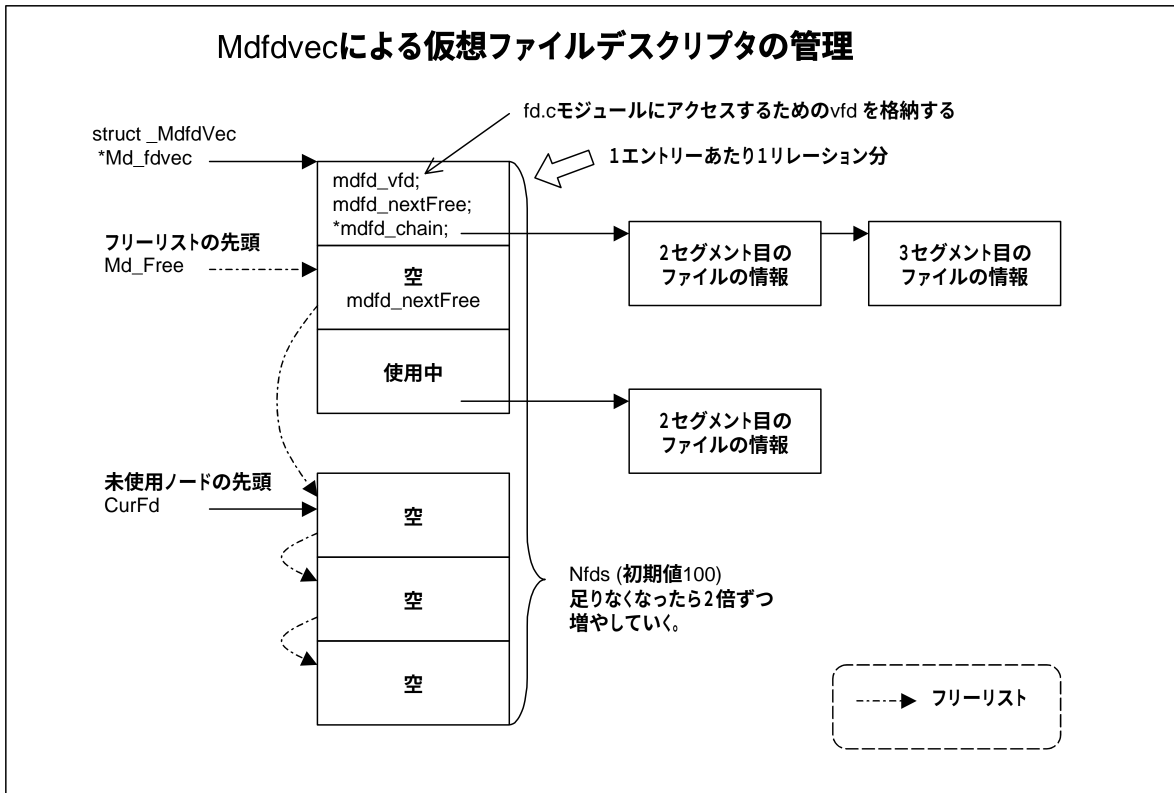

図 3

Md_fdvec 配列の１つのエントリが、リレーション１つに割り当てられる。このエントリが、リレーションの情報に保存されているファイルデスクリプタの値になっている。１つのリレーションが複数ファイルにまたがる場合、mdfd_chain リストに２番目以降のセグメントの情報が書き込まれる。２セグメント目以降のリストのノードは、フリーリストから使うのではなく、毎回アロケートしている。

物理ファイルの読み書きを行うための下位のインタフェースとしては、fd.c で定義されている関数を使用する。

2.5. mm.c

テーブルをメインメモリ上に格納するときに使用するモジュールである。メモリ DB としての動作をさせるためのものだと思われる。しかし、デフォルトでは、ブロック数 10、テーブル数 2 までしか使えないようになっているので、実用性がない。

```
#define MMNBUFFERS      10
#define MMNRELATIONS    2
```

データブロックを格納する構造は、次の図のようになっている。
データブロックは、データベースのＩＤ、リレーションのＩＤ、ブロック番号で一意に表すことができる。目的のデータブロックをすばやく見つけるために、これをキーにしたハッシュ MMCacheHT を用いる。

Copyright © 2003 NTTDATA

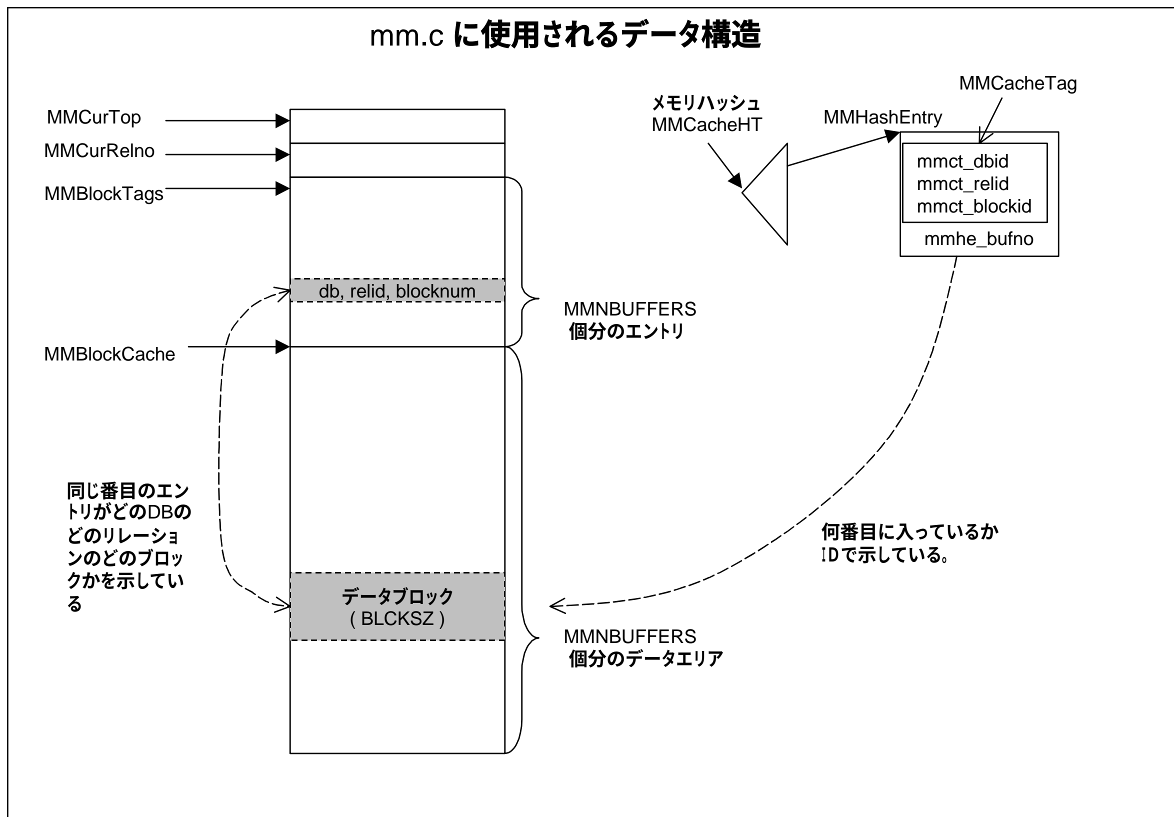

図 4

このほかに、１つのリレーションに何ブロック割り当てたかを管理するハッシュ MMRelCacheHT を持っている。ハッシュキーは、MMRelTag 構造体(データベースのＩＤとリレーションのＩＤの組)である。

3. storage/file

storage/file 中には、２つの機能が実装されている。１つは、OS のファイルデスクリプタを複数ファイルで共有するための仕組みを提供する fd.c である。もう１つは、一時ファイルを簡単に扱うために、stdio の fopen(), fread(), fwrite(), fseek(), fclose() 相当の機能を実装した buffile.c である。

3.1. 仮想ファイルデスクリプタ (fd.c)

fd.c は、仮想ファイルデスクリプタ「virtual file descriptors (VFD)」を管理するモジュールである。ただし、リレーションに割り当てられているファイルデスクリプタは、storage/smgr/md.c で管理されている Md_fdvec 配列のエントリなので注意が必要である。
VFD を使用した、ファイルアクセスのインタフェースとは、ＯＳのシステムコールの open(), read(),write(), close() 相当のものである。

ファイルを次々にオープンすると、システムのファイルデスクリプタが不足するため、ファイルデスクリプタを仮想化して fd.c によって実際にオープンするファイルの数をコントロールする。
PostgreSQL でのファイルオープンの主な要因は、テーブルのファイル、ソートやハッシュスプールなどに使う一時ファイル、system(3) などの C ライブラリルーチンの呼び出しである。これらを利用していると、すぐに OS のファイルデスクリプタの上限になってしまう。fd.c では、OS ファイルデスク

Copyright © 2003 NTTDATA

リプタを管理し、ファイルデスクリプタが不足した場合、他のファイルを閉じて、OS ファイルデスクリプタを割り当てなおす。仮想ファイルデスクリプタを使ってファイルにアクセスしたときに、割り当てられている OS ファイルデスクリプタが閉じられていたら、再度 OS レベルでファイルを開き、OS ファイルデスクリプタを割り当てる。VFD は、LRU アルゴリズムを用いて管理されている。

ダイナミックローダー(LD)もファイルデスクリプタを必要とする。しかし、VFD では、ダイナミックローダーの分のファイルデスクリプタまでは管理できないので、その分は予約として空けてある。

```
#define RESERVE_FOR_LD  10
```

ファイルデスクリプタを管理する都合、fopen() で開くファイルも、数だけ管理している。こちらについては、fopen(), fclose() 相当のインタフェースだけ用意してあり、fread() などは通常のシステムコールを使用する。

3.1.1. ファイル操作のインタフェース

fd.c には、外部インタフェースとして次のようなものを用意してある。

FileNameOpenFile	ファイル名(パス無し)で、ファイルをオープンする。 $PGDATA/base/... 以下のファイルを開くときに使用する。
PathNameOpenFile	フルパスでファイルを開く
OpenTemporaryFile	一時ファイルをオープンする。VFD で、一時ファイルであるフラグを立てることで、クローズ時に削除する。
FileClose	ファイルをクローズする。一時ファイルならファイルを削除する。
FileRead	ファイルの読み込みを行う
FileWrite	ファイルの書き出しを行う
FileSeek	ファイルのシークを行う
FileUnlink	強制的にファイルを削除する
AllocateFile	fopen() でファイルを開きたい場合に使用する関数
FreeFile	AllocateFile() で開いたファイルと閉じる

※ このほかに、BasicOpenFile() という内部の関数がある。通常の場合、BasicOpenFile()を直接呼び出すべきではないのだが、すぐに閉じることがわかっているファイルを開く場合、これを利用することも可能である。通常の open() システムコールと違って、ファイルデスクリプタが不足している場合に、他のファイルを閉じてファイルを開くことまでやってくれる。ただし、BasicOpenFile() を呼んだだけでは、そのファイルデスクリプタは再利用されないので、利用するのは限られた場合だけにするべきである。実際に、この使い方もソースコード中で見かけることがある。

Copyright © 2003 NTTDATA

3.1.2. 内部関数

仮想ファイルデスクリプタを管理するための内部関数は次のようなものがある。

Delete	ファイルを LRU のリストから削除する
LruDelete	ファイルを LRU のリストから削除し、ファイルを close する
Insert	ファイルを LRU のリストの先頭に入れる
LruInsert	ファイルを LRU のリストの先頭に入れて、ファイルを open する
ReleaseLruFile	LRU の最後尾のエントリのファイルをクローズして、OS のファイルデスクリプタを１つ空ける
AllocateVfd	空の Vfd のエントリを Vfd 配列から割り当てる
FreeVfd	Vfd のエントリを開放する
FileAccess	ファイルアクセス時に LRU の順序を先頭に移動させる

3.1.3. VFD を管理するデータ構造

VFD の管理には次のような構造体が用いられている。

```
typedef struct vfd
{
    signed short fd;            /* 現在割り当てている OS の fd、close してある場合は-1 */
    unsigned short fdstate;     /* VFD の状態を表すビットフラグ */

    File        nextFree;       /* フリーリスト。次の free の VFD へのポインタ */
    File        lruMoreRecently;        /* LRU の双方向リスト。先頭側へのポインタ */
    File        lruLessRecently;        /* LRU の双方向リスト。後方側へのポインタ */
    long        seekPos;        /* ファイル中の位置。再オープンした際に正しい位置に
                                   seek するために必要 */
    char       *fileName;       /* 実ファイル名。NULL ならこの VFD は未使用 */
    int         fileFlags;      /* open でファイルをオープンするときに使用するフラグ */
    int         fileMode;       /* open でファイルをオープンするときに使用するモード */
} Vfd;

◇ 実体へのエントリポイント
static Vfd *VfdCache;
static Size SizeVfdCache = 0;
```

VFD は、ファイルオープンの要求の数だけ作成されるのに対して、ＬＲＵは実際にＯＳレベルでオープンしているファイルに対してのみ使用している。ここを押さえておくと、fd.c のソースは読み易いと思う。

実際のファイルがオープンされている場合、lruMoreRecently, lruLessRecently を使ったＬＲＵの双方向リストにつながれ、実ファイルが閉じられている場合、ＬＲＵのリストから外される。
VFD 自体が開放されたら、nextFree を使ったフリーリストにつながれる。

図５は、VfdCache のデータ領域の拡張され方、VFD のフリーリストの管理方法、ＬＲＵの管理方法

Copyright © 2003 NTTDATA

を図にしたものである。特徴としては、VfsCache は、fd 0 が stdin であることを利用し、VfdCache[0]を LRU 先頭と最後のポインタや、フリーリストの先頭を指すポインタとして利用している。

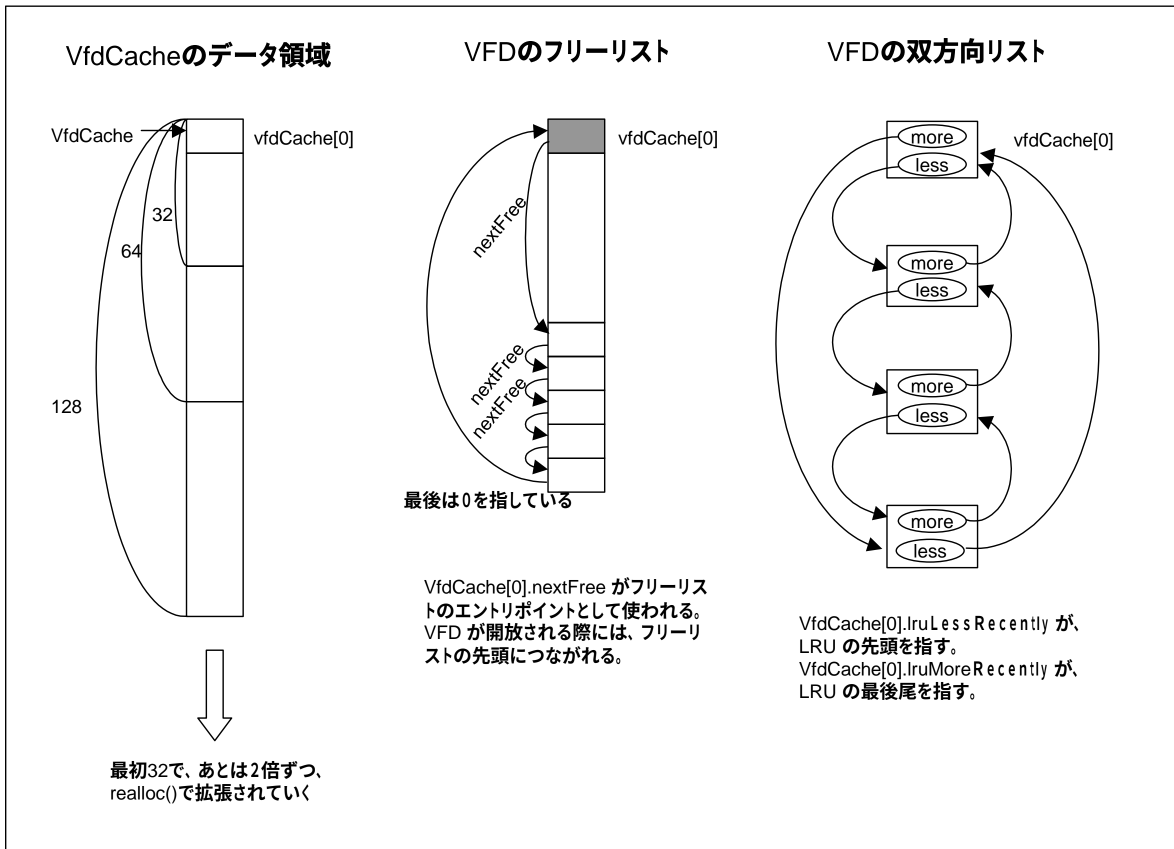

図 5

3.1.4. fopen() 系の I/O インタフェース

通常のルーチンでは、fopen() をそのまま使うのではなく、AllocateFile() を使わなければならない。AllocateFile()では、ファイルオープン時に OS のファイルデスクリプタが不足していたら、他のファイルを閉じることで、ファイルデスクリプタの開放を行う。AllocateFile() を呼んだ場合は、fclose() ではなく FreeFile() を呼ばなければならない。
インタフェースは用意してあるものの、この方法は、単に config ファイルを読み込みなど、読んですぐ閉じるものに限定しなければならない。長時間開くファイルをこの方法で開くと、カーネルのファイルデスクリプタを共有できなため、ファイルデスクリプタを使い切ってしまう危険がある[3]。

fd.c は、commit, abort 時に、AllocateFile()でオープンされた全てのファイルを自動的に閉じてくれる。これによって、AllocateFile()を呼び出して elog(ERROR)で終了した場合の、ファイルデスクリプタのリークを防ぐようになっている。

fopen() の管理をするデータ構造は、ファイルポインタを配列に格納しているだけである。次の図のように、配列の先頭から使っていき、新しいファイルを割り当てる際には、配列の利用している部分の

[3] そうは言っても上限が 32 個分にしてあるので、それで足りる範囲なら現実的には問題はない。

Copyright © 2003 NTTDATA

次のエントリを割り当てる。ファイルを開放するときは、該当エントリを削除したあと、空いた部分に一番後ろのエントリを持ってくることで、隙間を埋める。これによって、毎回必ず一番後ろの次を割り当てるというロジックが成立している。

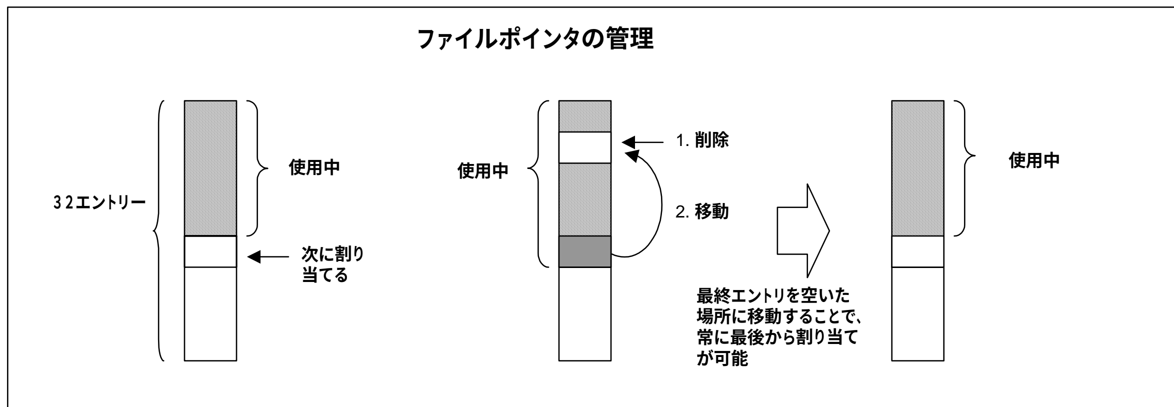

図 6

3.2. バッファファイル (buffile.c)

BufFiles は、fd.c の仮想ファイルデスクリプタの上に実装された、バッファ I/O の仕組みである。fread(), fwrite() 相当の可変長での I/O インタフェースを提供し、実 I/O をバッファ単位で行う。バッファとしては、メモリ上にブロックサイズ１つ分を使用するが、BufFiles で読み書きできるファイルサイズとは無関係である。単にファイル I/O を起こすサイズがバッファサイズの単位で行われるだけである。主に一時ファイルを扱うために使用される。

ファイルのサイズが大きくなりすぎると、OS のファイルサイズの限界に達してしまう場合がある。その場合、OS レベルのファイルを自動的に複数に分けて管理し、呼び出し側にそれを意識させないインタフェースを提供している。

BufFile 構造体は、palloc() を使用することで、トランザクション終了時に自動的に開放される。OpenTemporaryFile() でファイルを作成しておけば、elog(ERROR) でアボートしても、ファイル関連のリソースは開放される。混乱をさけるために、呼び出し側は、１つの BufFile は同じ palloc()の コンテキストから呼び出さなければならない。

3.2.1. 外部インタフェース

BufFile が提供している外部インタフェースは、次のようなものである。

BufFileCreateTemp	一時ファイルとして、BufFile を作成する
BufFileClose	BufFile のクローズ (終了処理)
BufFileRead	fread() とほとんど同じ。buffer サイズ単位でファイルに I/O を起こして ptr の先にデータをコピーする
BufFileWrite	fwrite() とほとんど同じ。バッファ単位で I/O を起こして、ptr のデータをファイルに書き出す

Copyright © 2003 NTTDATA

BufFileSeek	何番目のファイルのどの位置までという fseek が可能
BufFileSeekBlock	ブロック単位の seek を行う

永続ファイルに対するインタフェースは、#ifdef NOT_USED で囲まれて消されている。
BufFileRead()、BufFileWrite()時は、小さい単位で I/O を起こしても、バッファが終わりまで行かないと実 I/O は発生しないようになっている。

3.2.2. 内部関数

主な内部関数として次のようなものがある。

makeBufFile	新しい BufFile 構造体を作成する
extendBufFile	BufFile に新しい一時ファイルを１つ追加する
BufFileLoadBuffer	curOffset からバッファにファイルを読み込む
BufFileDumpBuffer	バッファをファイルに書き出す。書き出し位置は、curOffset
BufFileFlush	fflush のようなもの。dirty フラグが立っている場合、BufFileDumpBuffer(file); を強制的に実行する

実装上、一時ファイル以外も扱えるような雰囲気もあるが、しばらく手が入れられていないものと思われる。OS のファイルサイズの上限に対する対応がないなど、本当に使うならかなり手を入れる必要がある。

3.2.3. データ構造

現在位置 curOffset は、論理的なファイル中のバッファの始まる位置になる。BufFile の読み書き BufFileRead(), BufFileWrite()で使用する位置は、curFile と curOffset + pos で表すことができる。
バッファサイズと、BufFileRead(), BufFileWrite() の引数で使われるサイズは無関係である。単純に I/O を起こす単位がバッファのサイズ BLOCKSZ になるだけである。

Copyright © 2003 NTTDATA

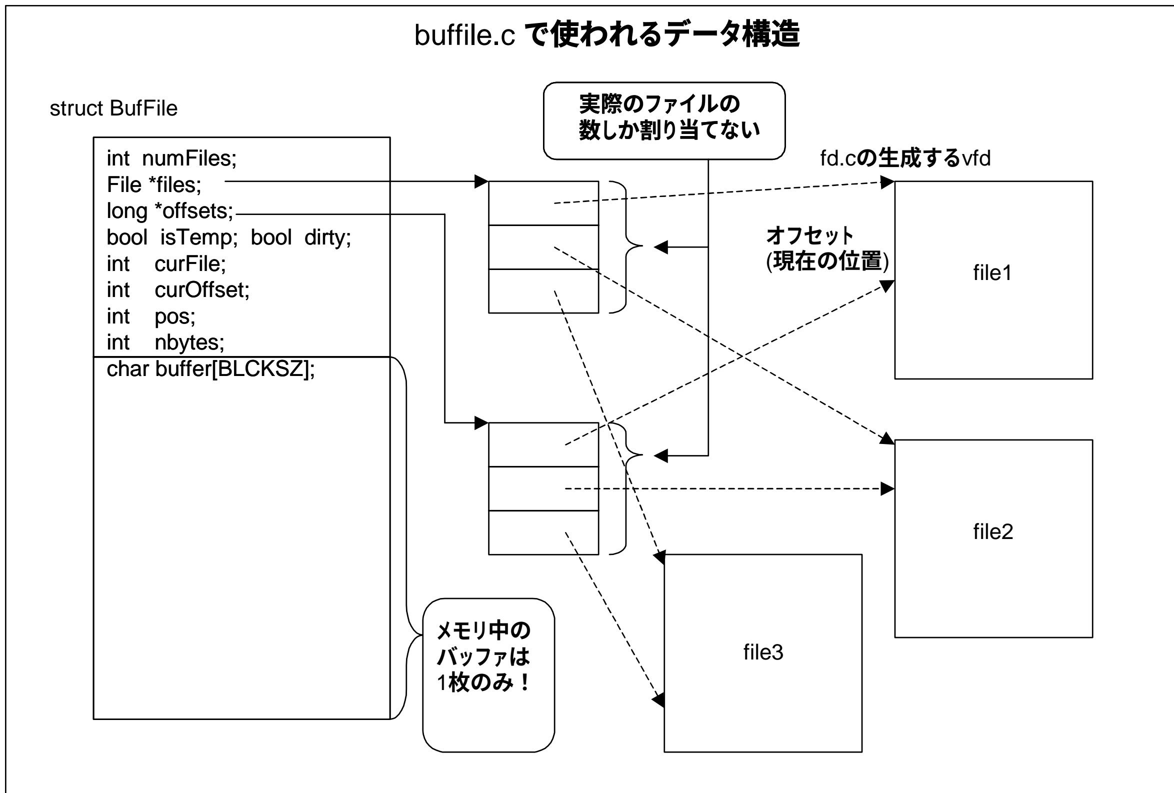

図 7

struct BufFile 中で、次の２つの先に割り当てられる領域は、ファイルの数に合わせて動的にサイズが変更される。

File	***files;**	**numFiles** 個の配列。仮想 **fd** が入っている。
long	***offsets;**	**BufFile** を構成する各ファイルの現在のオフセット

struct BufFile の中で、変数の意味がわかりづらいところを図にした。

Copyright © 2003 NTTDATA

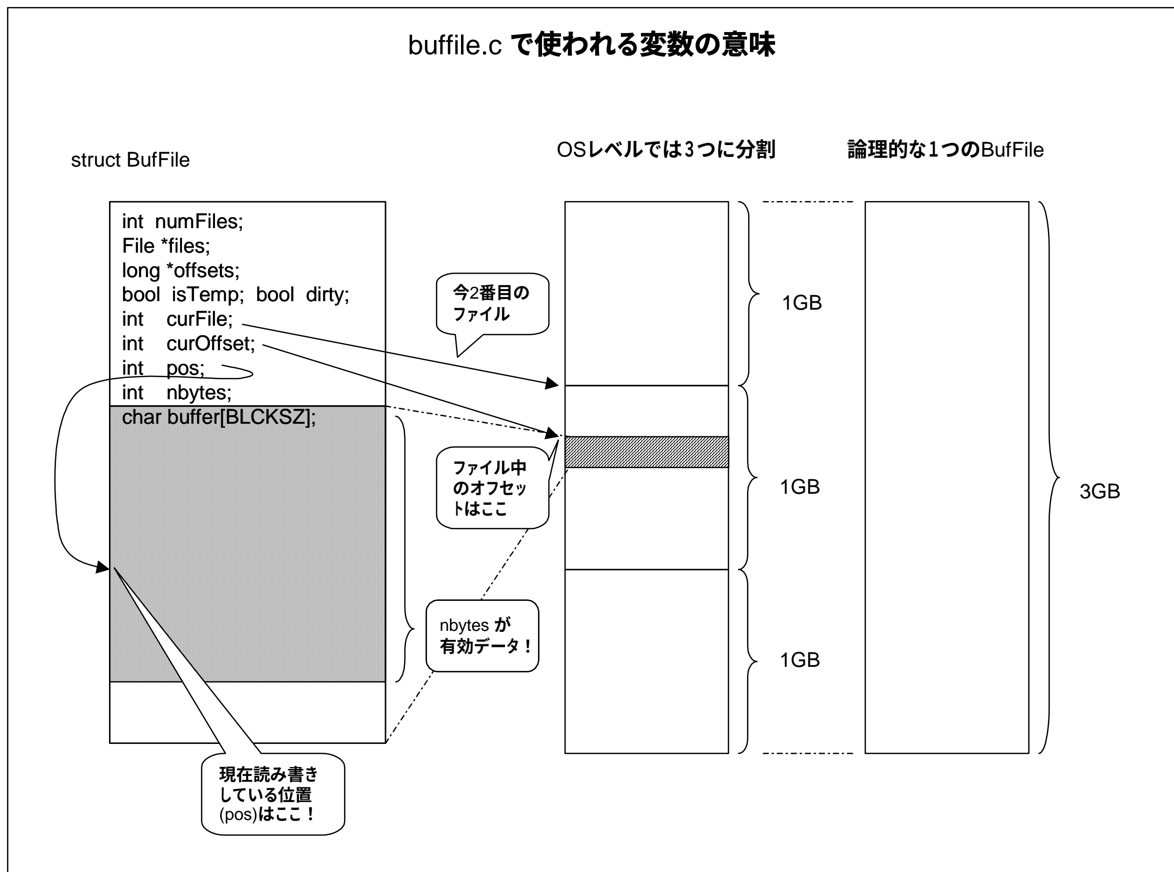

図 8

図 8 の中では、３ＧＢの BufFile を想定しているが、これは物理的には３つの 1GB のファイルで構成される。この図では、BufFile 構造体のバッファに、２番目のファイルの中ほどのデータを読み込んである状態である。BufFile 構造体の変数は、それぞれ次のようになっている。

- curFile は、今、何番目のファイルにアクセスしているかを表している。
- curOffset が現在アクセスしているファイル中のバッファに読み込んだ場所の先頭を示している。
- pos は、BufFileRead(), BufFileWrite()で読み書きした、バッファ中の位置である。
- nbytes は、バッファ中で有効なデータのサイズである。

ソースコードを読む際に、curOffset, pos, nbytes の動きが非常に複雑である。ここで、BufFileRead()時の動きを例に挙げて説明する。
次に示してあるのが buffile.c の BufFileRead() のコードである。

```
size_t
BufFileRead(BufFile *file, void *ptr, size_t size)
{
    size_t      nread = 0;
    size_t      nthistime;

    if (file->dirty)
    {
        if (BufFileFlush(file) != 0)
            return 0;           /* could not flush... */
```

Copyright © 2003 NTTDATA

```
        Assert(!file->dirty);
    }

    while (size > 0)
    {
        if (file->pos >= file->nbytes)
        {
            /* Try to load more data into buffer. */
            file->curOffset += file->pos;
            file->pos = 0;
            file->nbytes = 0;
            BufFileLoadBuffer(file);
            if (file->nbytes <= 0)
                break;          /* no more data available */
        }

        nthistime = file->nbytes - file->pos;
        if (nthistime > size)
            nthistime = size;
        Assert(nthistime > 0);

        memcpy(ptr, file->buffer + file->pos, nthistime);

        file->pos += nthistime;
        ptr = (void *) ((char *) ptr + nthistime);
        size -= nthistime;
        nread += nthistime;
    }

    return nread;
}
```

このソースコードを例にして、curOffset, pos, nbytes の変化を次に示す。

Copyright © 2003 NTTDATA

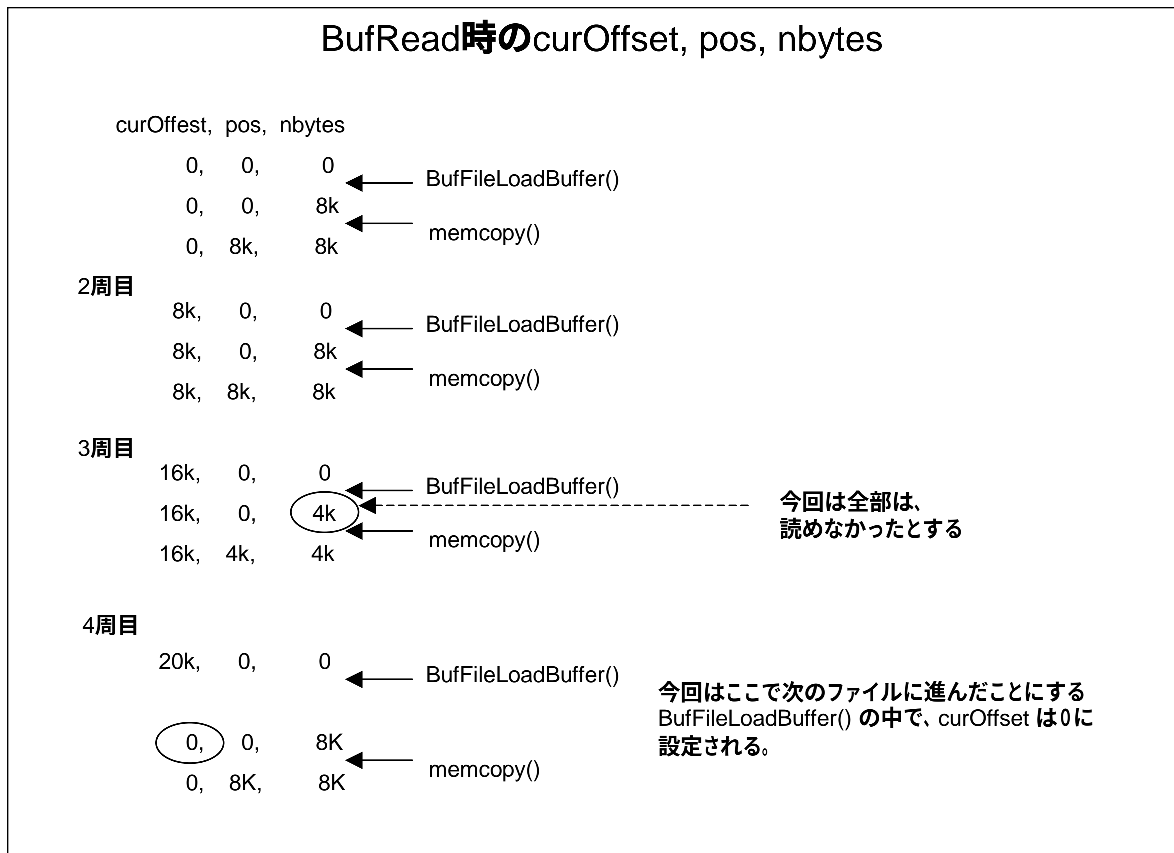

図 9

はじめに BufFile を開いた状態では、curOffset, pos, nbytes は、全て 0 である。１回目の BufFileLoadBuffer() を実行すると、バッファにデータが読み込まれた分の 8k が nbytes に設定される。次に memcopy() を実行したら、バッファ内のデータを 8k 消費したことになるので、バッファ内の位置である pos を 8 k 進める。

２周目のループで、curOffset が pos の 8k 進められ、pos、nbytes は、0 に初期化される。BufFileLoadBuffer() 以降の動きは、最初と同じである。

３周目でも、２周目同様、curOffset を進め、pos, nbytes は 0 にする。今度は、BufFileLoadBuffer() で、4k しか読めなかったことにする。読み込んだサイズの 4k が nbytes に設定される。この例のように、先頭から読み始めた場合はあり得ないが、BufFileSeek() などで位置を変えると、ファイルの末尾などでこのようなケースも考えられる。memcopy() でも 4k しかデータを使えないので、 pos が 4k になる。

４周目の先頭では、curOffset に pos の 4k が追加される。今度は、BufFileLoadBuffer() で、ファイルが次のファイルに進んだことにする。ファイルが変わるので、curOffset は 0 に設定されてくる。以降は、１周目、２周目のループと同じである。

Copyright © 2003 NTTDATA