PostgreSQL 解析資料

～ プロセス間通信（IPC） ～

(株) NTT データ
オープンソース開発センタ
井久保 寛明

1. はじめに

本ドキュメントでは、PostgreSQLで使われるプロセス間通信（IPC）に関連するモジュールについて説明している。主な内容は、共有メモリ、シグナルの拡張、共有inval メッセージ[1]についてである。共有 inval メッセージのモジュールからバックエンドプロセス管理の構造体にアクセスできるようになっているので、その仕組みも合わせて紹介する。

1.1. 対象バージョン

本ドキュメントは、PostgreSQL7.4.3 を対象にソースコードの調査を行ったものである。従って、他のバージョンでは、内容が異なる場合があるので注意して頂きたい。

1.2. ソースファイル

このドキュメントで対象にしているのは、src/backend/storage/ipc ディレクトリにあるファイルである。ここには、次のようなソースファイルがある。

ipc.c	postmaster や backend プロセスが、exit 時に行うクリーンナップ処理を補助する機能の実装。
ipci.c	プロセス間通信の初期化のコード。共有メモリの初期化関数 CreateSharedMemoryAndSemaphores() が定義してあり、その中から共有メモリを使用する各モジュールの初期化関数を呼び出す。
pmsignal.c	拡張シグナルの実装。
shmem.c	PostgreSQL で使用する共有メモリを管理するためのモジュール。
shmqueue.c	共有メモリで、双方向リンクを管理するパッケージ。通常の双方向リンクでは、ポインタを使うところを、オフセットで実現するためのもの。
sinval.c	共有 inval メッセージのインタフェースの実装と PGPROC 構造体をスキャンするために必要な関数群。
sinvaladt.c	共有 inval メッセージの管理構造を扱うモジュール。

[1] Shared-Invalidation Message。システムカタログやリレーションのキャッシュが無効になったことを伝えるメッセージ機構。

2. 共有メモリ

PostgreSQL では、バックエンドプロセス間でデータを共有するために、共有メモリを使用している。共有メモリは、postmaster 起動時にまとめて１つの領域として確保される。そして、この１つの共有メモリの領域をセグメントに切り分けて、様々な目的に使用する。この共有メモリの管理を行うのが shmem.c である。

2.1. アドレスとオフセット

共有メモリを使用する場合、あるプロセスで共有メモリを作成し、ほかのプロセスから同じ共有メモリをアタッチした場合、プロセス毎にアタッチされるアドレスが異なるため、共有メモリ中にポインタを保存することはできない。そのため、ポインタが必要な場合は、ポインタの指す先のアドレスを、共有メモリの開始アドレスからのオフセットとして保存する。
ほとんどのプラットフォームにおいて、PostgreSQLはバックエンドプロセスをpostmasterからforkして作成するため、postmasterで作成した共有メモリのアドレスとバックエンドに割り当てられている共有メモリのアドレスは一致している。従って、このようなオフセットを使用したアドレス指定をする必要はない。しかし、PostgreSQLのソースコードでは、次のようなマクロを定義して、アドレスとオフセットの変換を行っている。このようなコードになっている理由までは調べていないが、結果として移植性の高いコードであると言える[2]。

```
/* coerce an offset into a pointer in this process's address space */
#define MAKE_PTR(xx_offs)  (ShmemBase+((unsigned long)(xx_offs)))

/* coerce a pointer into a shmem offset */
#define MAKE_OFFSET(xx_ptr)  ((SHMEM_OFFSET) (((unsigned long)(xx_ptr))-ShmemBase))
```

ここで、ShmemBase には、各バックエンドプロセスが OS によって割り当てられた共有メモリの開始アドレスが入っている。MAKE_PTR()では、xx_offs として共有メモリのオフセットを与えると、そのアドレスのポインタを返す。また、MAKE_OFFSET()では、xx_ptr としてメモリアドレスのポインタを与えると、共有メモリ中でのオフセットに変換される。これらは、共有メモリが１つのまとまった領域として確保してあるため、このような変換が可能である。

このマクロを使って、ポインタをオフセットに変換する場合の使い方は次のようになる。

```
uint32 location;
void *structPtr;

location = MAKE_OFFSET(structPtr);
```

そして、マクロを使って、オフセットをポインタに戻す場合は次のようになる。

```
uint32      newStart;
void       *newSpace;
```

[2] Windowsのように、forkシステムコールのないようなプラットフォームに移植する際に役立つはずである。

```
newSpace = (void *) MAKE_PTR(newStart);
```

2.2. 共有メモリとセマフォの確保

共有メモリとセマフォを初期化する関数は CreateSharedMemoryAndSemaphores() である。この中の処理の概要は、次の図のようになる。

```
CreateSharedMemoryAndSemaphores()
  ├ [リソースの見積もり]
  ├ PGSharedMemoryCreate()
  ├ [セマフォ数の計算]
  ├ PGReserveSemaphores()
  ├ InitShmemAllocation()
  ├ CreateLWLocks()
  ├ InitShmemIndex()
  │   ├ ShmemInitHash()
  │   │   ├ ShmemInitStruct()
  │   │   │   ├ ShmemAlloc() <-- if (!ShmemIndex) の条件でこれだけ実行される
  │   │   ├ hash_create()
  ├ [各種モジュールの初期化]
```

図 2-1 共有メモリをセマフォの初期化関数の処理概要

まず、PostgreSQL の各モジュールで使用する共有メモリの量を見積もって、PGSheredMemoryCreate()で共有メモリを確保する。次に、必要なセマフォの数を計算して、PGReserveSemaphores()でセマフォを確保する。
続いて、共有メモリ内を初期化するために InitShmemAllocation() を呼び出す。InitShmemIndex() を呼び出す前に、LW ロックの初期化を行っておかなければならないので、CreateLWLocks()を先に実行している。
これらの処理が終わると、共有メモリを使用する各モジュールの初期化関数を順番に注意しながら呼び出す。

2.2.1. 共有メモリの確保を実装しているソースコード

共有メモリの確保は、storage/ipc/ipci.c の中の CreateSharedMemoryAndSemaphores() から行われる。
実際に共有メモリを確保するのは PGSharedMemoryCreate() であり、セマフォを確保するのは PGReserveSemaphores()である。
これらの関数は、それぞれ src/backend/port/pg_shmem.c と src/backend/port/pg_sema.c に定義されている。しかし、これらのファイルはシンボリックリンクになっている。共有メモリとセマフォは OS に依存するので、ソースコードをコンパイルする際の configure 実行時に、適切なプラットフォームのファイルにシンボリックリンクしている。Linux の場合は、それぞれ port/sysv_shmem.c と port/sysv_sema.c が使われる。

2.2.2. 共有メモリのサイズ

共有メモリのサイズ計算は、各モジュールで実装されている共有メモリの使用量の計算関数を呼び出して算出する。関連しているモジュールは次のとおりである。

ファイル名	サイズ見積もり関数	概要
storage/buffer/buf_init.c	BufferShmemSize()	Shmem Indexハッシュテーブルのサイズ、バッファデスクリプタのサイズ、データバッファのサイズ[3]、バッファ用ハッシュテーブルのサイズの合計
storage/lmgr/lock.c	LockShmemSize(maxBackends)	プロセステーブルのヘッダ、プロセステーブルのエントリ、ロックメソッドテーブル、ロックのハッシュテーブル、プロセスロックのハッシュテーブルの合計に 10%を加算したサイズ。
access/transam/xlog.c	XLOGShmemSize()	xlog のコントロールデータ、xlog のバッファ、xlog のバッファのヘッダ領域、pg_control 用の領域の合計。
access/transam/clog.c [access/transam/slru.c]	CLOGShmemSize() [SimpleLruShmemSize()]	CLOGShmemSize() は、単純に SimpleLruShmemSize() を呼び出す。 Slru のヘッダと CLOG バッファのサイズの合計。
storage/lmgr/lwlock.c	LWLockShmemSize()	LW ロックの数の分のロック領域と int ２つ分の領域。LW ロックの数は、ロックの種類(現在 13 種類)＋バッファ数の２倍＋CLOG バッファ＋１個である。
storage/ipc/sinvaladt.c	SInvalShmemSize(maxBackends)	共有 inval セグメント(ヘッダ)とプロセスステータス構造体がバックエンドの数分のデータ量。
storage/freespace/freespace.c	FreeSpaceShmemSize()	FSM のヘッダと FSM リレーション用のハッシュテーブル、ページ情報格納用のチャンクの合計。

各モジュールでは、共有メモリを使う場合、固定サイズでセグメントを確保しなければならない。可変長のデータ構造を作成する場合は、必要なメモリの最大サイズを計算した上で共有メモリのセグメントを確保し、そのセグメント内で調整することになる。

共有メモリ全体のサイズは、これらの関数の戻り値の合計に、100KB を加えたものを 8192 で丸めたサイズになっている。
例えば storage/ipc/pmsignal.c ように、上の見積もりテーブルに入っていないのに、少量の共有メモリ

[3] 設定ファイルのパラメータshared_buffers ×ブロックサイズで計算されるサイズは、この部分だけである。

を確保しているものもあるため、100KB の余裕が必要となっている。

実際にこれらの合計値を基にすると、必要な共有メモリサイズを算出することも可能である。しかし、式が複雑なことと、プラットフォームやコンパイル時のオプションなどにより異なってくる場合がある。共有メモリの合計値を算出する一番お手軽な方法は、メッセージ表示レベル（log_min_messages）をデバッグの３以上にすることである。すると、デバッグメッセージの１つとして、次のように、確保された共有メモリのサイズが表示される。

```
DEBUG:  invoking IpcMemoryCreate(size=10436608)
```

2.2.3. セマフォの数

セマフォは、あるバックエンドプロセスがロック待ちになった場合に、他のバックエンドプロセスがロックを開放するのを検出する場合に使う。そこで、セマフォは最低でもバックエンドの数+1 が必要である[4]。

また、セマフォはTest And Set命令[5]がないプラットフォームの場合、スピンロックの実装にも用いている。従って、スピンロック用のセマフォが必要なプラットフォームもある。スピンロック用に使うセマフォは、LWロックの数 ＋ 10 個である。LWロックの数は、「固定のLWロック」と「共有バッファの２倍」と「CLOGのバッファ数＋１」を加えたものである。

Linux の場合は、Test And Set 命令があるので、スピンロック用のセマフォは必要ない。

2.2.4. 共有メモリの初期化と割り当て （データ構造）

共有メモリは、１つの領域として OS から割り当てられる。その後、その領域を初期化し、共有メモリを分割して目的毎に利用できるようにする。共有メモリ内の構造は、図 2-2 のようになっている。

[4] +1 はダミープロセス用。

[5] mutexロックを実現するためのハードウェア命令。１命令であるアドレスのメモリの値を読むと同時に、別な値を保存する。これにより、ほかのスレッドに割込みによる競合が起こらないようにできる。

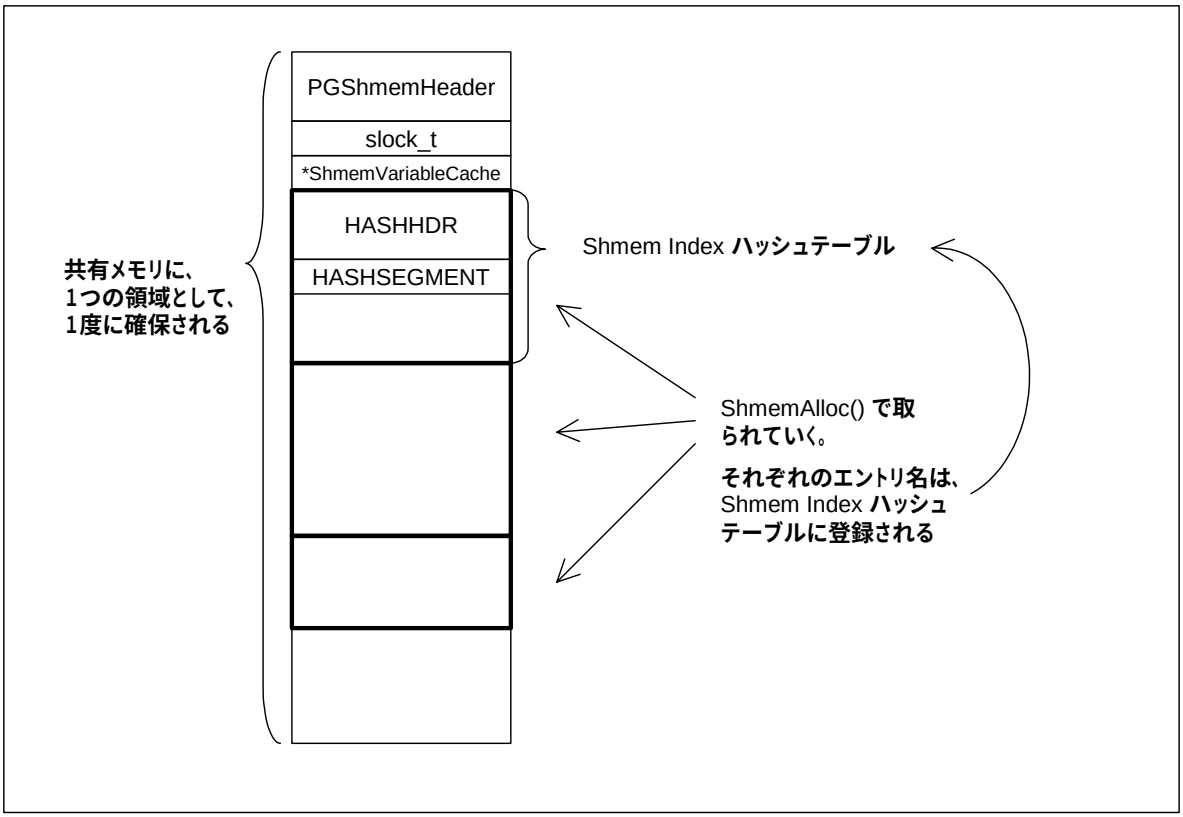

図 2-2 共有メモリのデータ構造

共有メモリの先頭にある共有メモリのヘッダ PGShmemHeader は、次のように非常に単純なものである。

```
typedef struct PGShmemHeader    /* standard header for all Postgres shmem */
{
    int32       magic;          /* magic # to identify Postgres segments */
#define PGShmemMagic  679834892
    pid_t       creatorPID;     /* PID of creating process */
    uint32      totalsize;      /* total size of segment */
    uint32      freeoffset;     /* offset to first free space */
} PGShmemHeader;
```

magic に、PostgreSQL が確保したセグメントであることを確認するマジックナンバーが入っている。続いて、**creatorPID** に共有メモリを作成したプロセスの ID を保管している。
あとは、共有メモリのセグメント全体のサイズ **totalsize** と、ShmemAlloc()で割り当てた、最後の位置を示す **freeoffset** である。

図 2-2 で、共有メモリのヘッダ PGShmemHeader の直後にある領域 **slock_t** は、共有メモリを管理する構造へのアクセスを排他制御するためのスピンロックに使用する領域である。次の
*ShmemVariableCache は、トランザクションマネージャで使用するポインタ用の領域である。今回は、これがどのように使われるかは調べていない。
これ以降が、共有メモリを目的毎に分割して利用するセグメントとなる。最初のセグメントは、次に説明する Shmem Index ハッシュテーブルである。

2.2.5. Shmem Index ハッシュテーブル

Shmem Index ハッシュテーブルは、OSから確保した共有メモリを分割して割り当てる共有メモリセグメントを管理するために使われる。共有メモリセグメントを確保する際は、名前をつけて確保しており、その名前がShmem Indexハッシュテーブルに登録される。従って、同じ目的で使用する共有メモリセグメントが既に存在する場合は、Shmem Indexハッシュテーブルを検索すると名前が見つかるので、その共有メモリセグメントにアタッチする[6]ことで同じ共有メモリセグメントを利用する。

Shmem Index ハッシュテーブルは、最初にアロケートされる共有メモリの領域なので、図 2-2 のようにヘッダなどを除くと、最初の共有メモリセグメントとして割り当てられる。Shmem Index ハッシュテーブルの名前自体も Shmem Index ハッシュテーブルに格納してある。

Shmem Indexハッシュテーブルの初期化は、図 2-1 のInitShmemIndex() の部分である。共有メモリのセグメントとしてハッシュ構造を作るインタフェースShmemInitHash() があるので、Shmem Indexハッシュテーブルも、それを使って構築する。その関数内で使用する共有メモリのセグメントを確保する関数ShmeminitStruct() が、Shmem Index ハッシュテーブルを使用するので、卵と鶏の問題が発生する。その部分は、ShmemIndex というグローバル変数がNULLかどうか見ることで、Shmem Index ハッシュテーブルの初期化処理中かどうか判断する[7]。

2.2.6. 各種モジュールの初期化

共有メモリの初期化の際に、PostgreSQL で共有メモリを使っている他のモジュールの初期化関数の呼び出しも行っている。各種モジュールの初期化では、次のような順で初期化を行う。

1. xlog、clog、共有バッファの初期化を行う
   XLOGShmemInit(); <access/transam/xlog.c>
   CLOGShmemInit(); <access/transam/clog.c>
   InitBufferPool(); <storage/buffer/buf_init.c>
2. lock manager のセットアップを行う
   (i) ロックモジュールの初期化として、ロックマネージャ内のデータ構造を作る
   InitLocks(); <storage/lmgr/lock.c>
   (ii) ロックテーブルの初期化を行う
   InitLockTable(maxBackends) <storage/lmgr/lock.c>
3. process table をセットアップする
   InitProcGlobal(maxBackends); <storage/lmgr/proc.c>
4. 共有 inval メッセージを初期化する
   CreateSharedInvalidationState(maxBackends); <storage/ipc/sinval.c>

[6] OSレベルでの共有メモリへのアタッチ相当の機能を、PostgreSQLの共有メモリ管理モジュールでも、このモジュールの提供する共有メモリセグメントに対して提供している。
[7] ソースコード中に static bool ShmemBootstrap という変数が定義されていて、これが初期化中の状態を表すように見えるのだが、この変数は使われていないようである。(Assertには使っている。Assertも、変数ShmemIndexがNULLかどうかの判定に置き換えが可能である。)

5. FSM（Free Space Map）をセットアップする
   InitFreeSpaceMap(); <storage/freespace/freespace.c>
6. 子プロセスから postmaster へのシグナル通知機構のセットアップを行う
   PMSignalInit(); <storage/ipc/pmsignal.c>

2.3. 共有メモリのインタフェース

共有メモリセグメント（OS から確保した領域の切り分けた部分領域）は、主に３つの種類のデータ構造として扱うことができる。１つは、フラットな領域をそのまま確保して使用する構造である。２つ目もフラットな構造であるが、名前を管理することで既に同じ目的のセグメントが存在した場合は、その領域をアタッチさせる構造である。３つ目は、共有メモリ中にハッシュ構造を作る方法である。

共有メモリセグメントを確保するためのインタフェースは、次のようなになっている。

ShmemAlloc()	１つの領域として OS から確保した共有メモリから、セグメントとして、要求されたサイズを割り当てる。割り当てる際にアラインメントしたサイズで割り当てる。アラインメントは通常４バイト。ただし、ブロックサイズより大きな割り当てに対しては３２バイトアラインメントを行っている。 共有メモリのセグメントを使い果たした場合、エラーになる。
ShmemInitStruct()	共有メモリに、名前で指定した領域の作成またはアタッチする。まだ、この名前の領域がなければ、新しく作成する。もし、すでにこの名前の領域が他のプロセスなどによって割り当てられていた場合は、引数の FoundPtr に true を返す。 戻り値は、新しく割り当てたにしても、既にあったものを見つけたにしても、名前で指定された領域へのポインタである。
ShmemInitHash()	共有メモリ中にハッシュテーブルを作成またはアタッチを行う。共有メモリ中のハッシュは、バケットに使用する最大サイズが決まっているので、それも指定する。

共有メモリは、DBMS 全体で使用するデータを保存するので、共有メモリセグメントを確保することはあっても、共有メモリセグメントを開放することはない。一度共有メモリセグメントを作成したら、あるプロセスが使わなくなってもそのままにしておき、他のプロセスから使うときにアタッチするという使い方をする。そのため、個別の共有メモリセグメントを開放するためのインタフェースは存在しない。また、共有メモリセグメントを確保する場合は、必ず固定長の領域として確保する。可変長のデータ構造で使用する場合は、想定する最大サイズで共有メモリセグメントを確保し、そのセグメント内で調整を行う。

2.3.1. ShmemInitStruct() の実装

この関数は、少々注意が必要である。この関数は、インタフェースとして共有メモリの領域を割り当てる以外にも、共有メモリ管理構造の初期化にも使用している。
まだ Shmem Index ハッシュテーブルがない場合、共有メモリ管理構造の初期化中とみなして、Shmem Index ハッシュテーブル用の領域を割り当ててリターンする。

Shmem Index ハッシュテーブルがあるなら、指定された名前が Shmem Index ハッシュテーブルにあるかどうか検索する。もし見つかったらその領域のポインタを返す。そのときは、FoundPtr は TRUE である。
ここで、Shmem Index ハッシュテーブルから見つからなかった場合は、新規に ShmemAollc()で共有メモリの領域を割り当てて、Shmem Index ハッシュテーブルに、名前と領域へのオフセットを登録する。そして、割り当てた領域へのポインタを返す。この場合は、FoundPtr は FALSE である。

2.3.2. ShmemInitHash() の実装

まず、これから作るハッシュの領域を確保、または、すでにそのハッシュがあるならアタッチするために、ShmemInitStruct() を呼び出す。ShmemInitStruct()を使用すれば、新しい領域にしても既にあった領域にしてもハッシュのための領域のポインタを得られる。既に領域があった場合は、引数に渡した found に TRUE が返ってくるので、ハッシュ用の関数へも、ハッシュへのアタッチであるため初期化が不要であることを伝えるフラグ（HASH_ATTACH）を設定する。
続いて、ハッシュテーブルに位置やサイズ情報、各種フラグなどを hash_create() 関数に渡すことで、ShmemInitHash() の実装は終わりである。

2.4. プロセス終了時のコールバック関数管理

共有メモリやセマフォは、バックエンドプロセスのライフサイクルよりも長い時間存在するため、バックエンドプロセスがエラーなどで終了する際に後始末をする必要がある。さもないと、バックエンドプロセスが終了した後に、共有メモリ中にゴミが残ってしまう。
この後処理を支援するために、PostgreSQL では、postmaster やバックエンドプロセスの終了時に実行する関数を登録するしくみが用意されている。
共有メモリなどにアクセスした際に、プロセスが終了するときにリソースを開放しておかなければならないものに対して、リソースを削除するコールバック関数を登録する。プロセスの終了時には、これらの登録された関数が、登録したときと逆順に実行されていく。図 2-3 は、共有メモリのコールバック関数を管理するデータ構造である。

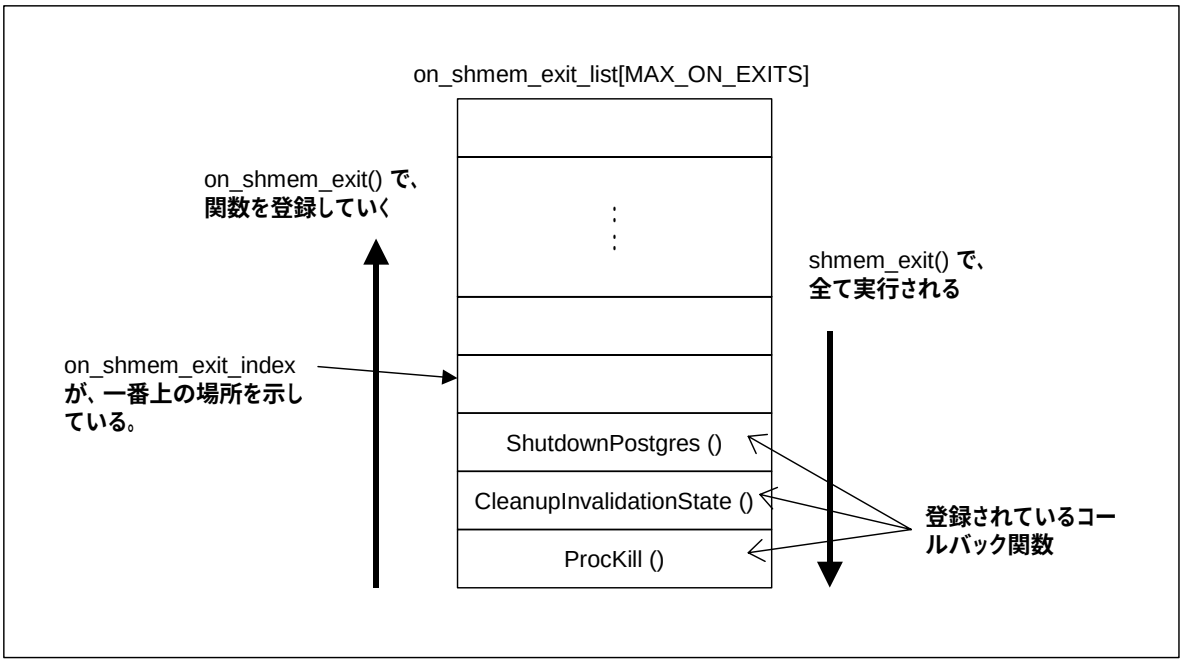

図 2-3 共有メモリのコールバック関数管理に使われるデータ構造

2.4.1. コールバック関数管理モジュールの実装

コールバック関数は、引数が 0, 1, 2 個のどれかである。１番目の引数には、終了時のint型のexit コードを指定できる。２番目の引数には、コールバック関数登録時に作成したDatum[8] が１つ指定できる。

また、コールバック関数は、共有メモリに関するものとプロセスの終了に関するものの２つに分けて管理される。これは、まず、共有メモリに関するコールバック関数を先に実行した後、各モジュールの終了のためのコールバック関数を実行したいという要望によるものである。

コールバック関数を管理するデータ構造は次のようになっている。

```
#define MAX_ON_EXITS 20

static struct ONEXIT
{
    void        (*function) ();
    Datum       arg;
}   on_proc_exit_list[MAX_ON_EXITS], on_shmem_exit_list[MAX_ON_EXITS];

static int  on_proc_exit_index, on_shmem_exit_index;
```

on_proc_exit_list と on_shmem_exit_list がそれぞれ 20 個ずつ登録できる。各エントリは、コールバック関数と Datum １つの組である。登録されている個数を管理するために、ローカル変数の on_proc_exit_index, on_shmem_exit_index がある。これらは個数と同時に次のエントリを示す。

[8] PostgreSQLで扱うデータ型は、様々なフォーマットがある。それを抽象的なデータ型として扱うのがDatumである。詳しくは、「初期DBの作成処理（Bootstrap）」の資料を参照。

2.4.2. インタフェース

コールバック関数のインタフェースは次のようなものがある。

関数	概要
proc_exit()	プロセスが終了するときに、登録されているコールバック関数を全て実行する関数。内部では、先に shmem_exit()を呼び出して、共有メモリに関連付けて登録されているコールバック関数を実行する。その後、プロセスの終了に関連付けて登録されているコールバック関数を順に実行する。
shmem_exit()	コールバック関数のうち、共有メモリに関連して登録されているコールバック関数を全て実行する関数。
on_proc_exit()	on_proc_exit_list 配列に、コールバック関数とその引数を登録する。
on_shmem_exit()	on_shmem_exit_list 配列に、コールバック関数とその引数を登録する。
on_exit_reset()	on_proc_exit_list 配列と on_shmem_exit_list 配列を空にする。

使い方は、まず、初期化のために on_exit_reset() を呼び出して、コールバック関数の初期化を行う。そして、コールバックによる後処理が必要なリソースを使った際に、on_proc_exit() または、on_shmem_exit() で、コールバック関数とその引数を登録する。
postmaster、または、バックエンドプロセスの終了時には、exit() システムコールの代わりに proc_exit() を呼ぶことで、on_proc_exit() または、on_shmem_exit() で登録されたコールバック関数が全て呼び出される。shmem_exit()は、proc_exit() から呼び出されるので、現状のソースコード上は、shmem_exit() だけを他のモジュールから呼び出すことはない。
on_exit_reset() による初期化は、まず、postmaster で行われる。postmaster から fork されたバックエンドプロセスは、postmaster 用のコールバック関数を実行する必要はないので、fork された時点でもう１度 on_exit_reset()を呼び出すことにより、バックエンドのコールバック関数だけを実行するように初期化する。

2.5. 共有メモリリンクリスト（shmqueue.c）

IPC 関連のモジュールの中に、共有メモリ中の双方向リストを管理するモジュールがある。それが、shmqueue.c であり、共有メモリリンクリストと呼ばれる。
双方向リストを管理するモジュールは、src/backend/nodes/link.c などもある。これらの一番大きな違いは、link.c では、リストをメモリアドレスのポインタで管理しているのに対して、shmqueue.c ではオフセットで管理する点である。
shmqueue.c では、双方向リンクのリストを実現しているが、キューを実現するのが目的のため、キューを実現することができる程度の簡単なインタフェースしか持たない。

shmqueue.c のインタフェースは次のようになっている。

SHMQueueInit()	新しいキューの先頭を作成する。

SHMQueueElemInit()	キューのエレメントをクリアする。
SHMQueueDelete()	リストからキューのエレメントを取り除く。
SHMQueueInsertBefore()	エレメントを、与えられたキューの直前に入れる。 先頭のキューの直前に入れることは、エレメントをキューの最後に入れたことになる。
SHMQueueNext()	キューから次のエレメントのポインタを返す。キューから削除は行わない。
SHMQueueEmpty()	キューの先頭だけしかエレメントがない場合、つまりキューが空の場合、TRUE を返す。その他の場合、つまりキューに何か入っていたら FALSE を返す。

キューには、初期化時に１つのエレメントを生成する。これが、キューの管理用のエレメントになる。つまり、キューが空の場合は１つのエレメントがある状態である。
キューは円形の双方向リストになるので、キューの管理エレメントの直前にエレメントを入れると、キューの一番最後に入れたことになる（図 2-4）。

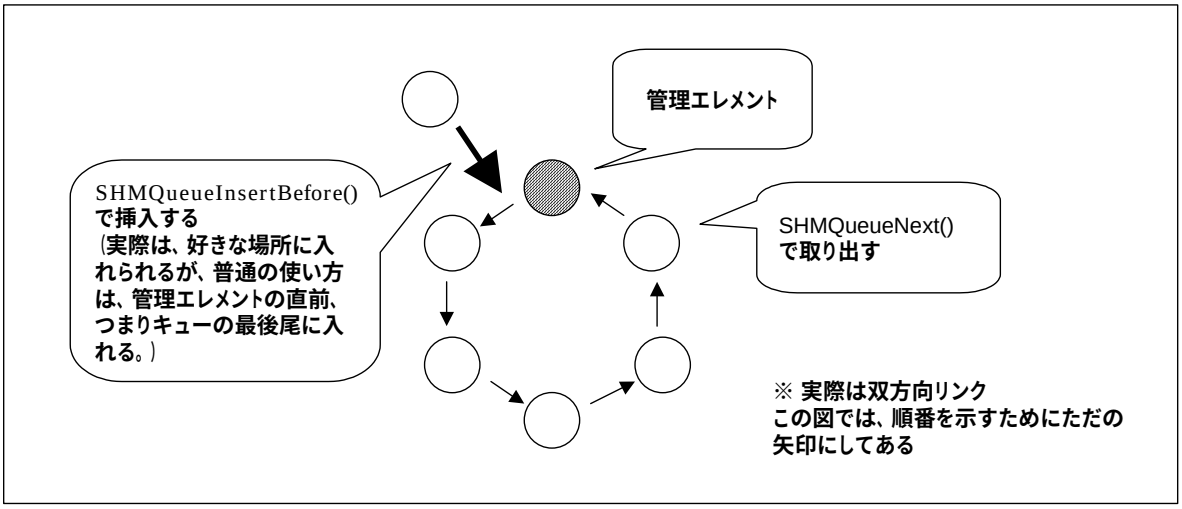

図 2-4 メモリリンクリストによるキューの管理

2.5.1. SHMQueueNext() によるエレメントの取り出し

キューを管理するための構造 SHMQueue を、実際に管理したい構造体の中に埋め込んで使うのが普通である。多くの双方向リンクの実装では、他の構造体に埋め込むと、キューからエレメントを取り出したときには、SHMQueue 構造体の位置のポインタが返ってくる実装が多い。結局呼び出し元で、SHMQueue の位置のオフセット分だけポインタを戻して、全体の構造体の先頭を算出する必要がある。または、SHMQueue 構造体を必ず全体の構造体の先頭に定義する必要がある。これでは、不便だと感じたのか、SHMQueueNext()で取り出すポインタは、全体の構造体の中の SHMQueue 構造体の位置ではなく、全体の構造体の先頭を返してくれるようにできている（図 2-5）。

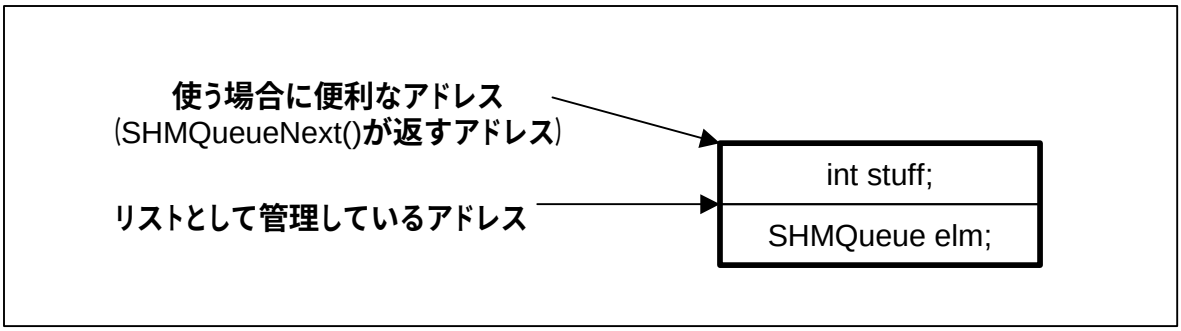

図 2-5 SHMQueueNext() が返すアドレス

例えば、次の構造体を定義したとする。

```
struct {
   int             stuff;
   SHMQueue        elem;
} ELEMType;
```

キューで管理しているのは、ELEMType.elem の位置である。SHMQueueNext() では、構造体のオフセットを渡すことで、この構造体の先頭を返してくれる。
実際に SHMQueueNext() に渡される引数は次のようなものである。

```
ptr = SHMQueueNext( &(queueHead), &(queueHead), offsetof(ELEMType, elem) )

ptr = SHMQueueNext( &(queueHead), &(curElem->elem), offsetof(ELEMType, elem) )
```

前者がキューの先頭のエレメントを返して欲しい場合で、後者が順番にキューの中を見ていく場合などに使用される。
第３引数の offsetof は、ELEMType 構造体の中で SHMQueue 構造体の elem が開始するオフセットが取り出せる。offsetof は、多くの OS では、stddef.h にマクロが定義されているが、一部の OS では存在しないので PostgreSQL でも次のように定義されている。

```
#define offsetof(type, field)   ((long) &((type *)0)->field)
```

3. シグナルの拡張

UNIX のシグナルは、32 ビット分のシグナルを全て定義してあり、その中でユーザ定義できるものは２つ（SIGUSR1, SIGUSR2）しかない。PostgreSQL では、バックエンドプロセスから postmaster プロセスに、もっと多くのシグナルを通知するためのしくみが入っている。

シグナルの拡張を実装するために、共有メモリに拡張シグナル用のエリアが用意してある。バックエンドから postmaster に拡張されたシグナル情報を伝える場合、まず、このエリアに、シグナル通知理由のビットを設定する。続いて、postmaster にシグナル SIGUSR1 を送る。シグナル SIGUSR1 を受け取った postmaster は、シグナルハンドラの中で、共有メモリの拡張シグナルエリアを調べ、シグナ

ルで伝えたかったことの内容を知ることができる。

PostgreSQL7.4.3 現在では、拡張シグナルで伝えることのできる内容は、次の３種類である。

シグナル理由	内容
PMSIGNAL_DO_CHECKPOINT	チェックポイントの実行を要求する。
PMSIGNAL_PASSWORD_CHANGE	pg_pwd ファイルが変更されたことを伝える。
PMSIGNAL_WAKEN_CHILDREN	NOTIFY シグナルを全てのバックエンドに送るように要求する。

3.1. 実装

それぞれシグナルの理由に対する「シグナル通知理由」用の boolean のフラグを持っている。それで、同時に別のバックエンドが発信した、別の「理由」を受け取ることができる。しかし、同じ理由が同時に複数通知されても、postmaster は１回しか受け取れない。現時点では、１回受け取ればいいものしか存在しないので、特に問題はない。仮に同時に同じリクエストが来たとしても、実行したいことは同じ内容なので、１回実行されればよい。

シグナル通知理由フラグは、最大のポータビリティのため、"volatile sig_atomic_t" として宣言される。これは、フラグの値の読み書きがアトミックになることを保証しているので、明示的なロックを省略できる。

```
static volatile sig_atomic_t *PMSignalFlags;
```

3.2. インターフェース

インターフェースは非常に単純で、次の３つである。

PMSignalInit(void)	拡張シグナルに使う共有メモリ領域の割り当てと初期化を行う。
SendPostmasterSignal(reason)	バックエンドプロセスから postmaster にシグナルを送る。内部で、理由フラグを設定して、SIGUSR1 を発行する。
CheckPostmasterSignal(reason)	個別のシグナル理由のフラグをチェックして、フラグが立っていたらクリアする。フラグが立っていたかどうかで true か false を返す。この関数は、postmaster で SIGUSR1 を受け取ってから呼び出されなければならない。

postmaster で SIGUSR1 を受け取ったら、それぞれの理由に対して CheckPostmasterSignal() を実行し、フラグが立っていたらそれに対応する処理を行う。

4. 共有 inval メッセージとプロセス構造体

PostgreSQL では、ユーザ定義型やユーザ定義の演算などが扱えるという非常に柔軟な構造を持っている。これらの情報は、システムカタログに格納されることで実現されている。PostgreSQL では、システムカタログに、ユーザ定義の部分だけでなく、基本的なデータ型や演算の情報まで全て格納してあり、統一的な仕組みで扱うことができる。そのため、システムカタログへのアクセスが非常に多くなることから、システムカタログの情報は個々のバックエンドプロセスでキャッシュしている。これがカタログキャッシュである。もちろん共有メモリに格納して共有する方法もある。しかし、トランザクション実行中にカタログ情報が突然変更されると困るため、共有メモリにアクセスする際にロックが必要となる。トランザクション実行中という長時間に渡ってロックを確保すると、同時実行性能が著しく悪くなるので、共有メモリでキャッシュすることは行っていない。
また、リレーションの情報についても非常にアクセスが多いことから、リレーションのデスクリプタをキャッシュする方法として、リレーションキャッシュを実装している。
これら２つのローカルキャッシュに、キャッシュの情報が古くなったことを伝えるしくみが、共有 inval メッセージである（図 4-1）。

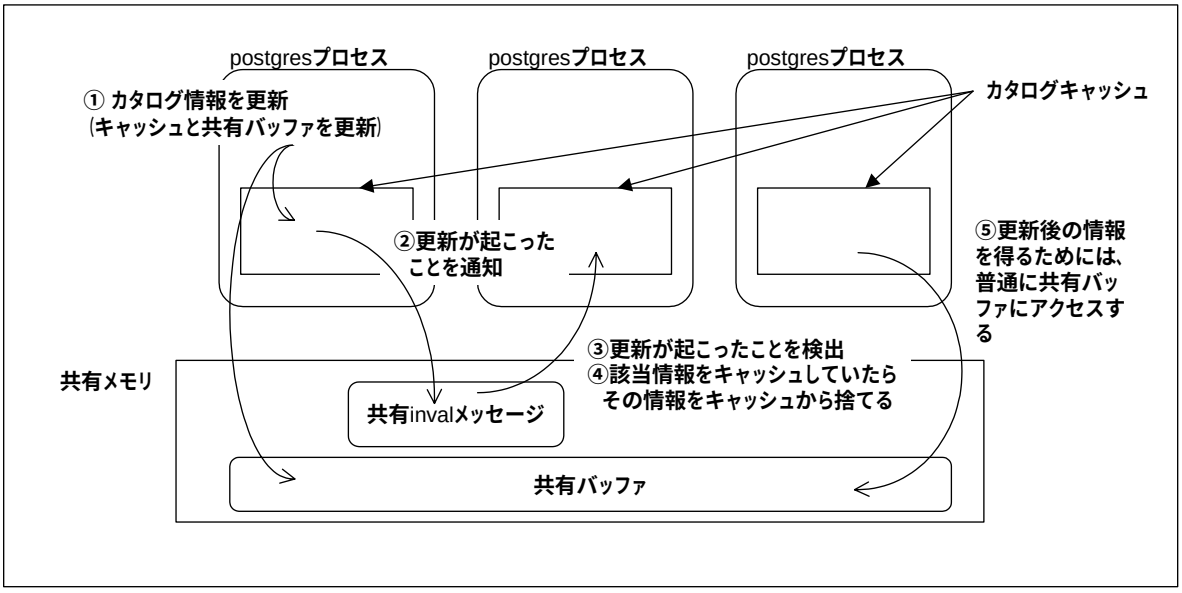

図 4-1 カタログキャッシュと共有 inval メッセージ

基本的な考え方は、カタログキャッシュかリレーションキャッシュの変更を行う場合、そのバックエンドプロセスはカタログやリレーションの情報の更新を行う。そのときに、更新対象のオブジェクトをキャッシュしている他のバックエンドに、キャッシュが失効したこと（無効になったこと）を伝えるメッセージを作成し、共有 inval メッセージのキューに登録する。例えば、リレーションを更新する場合は、そのリレーションのＩＤを含んだメッセージを登録する。他のバックエンドプロセスは、このオブジェクトのキャッシュの情報を更新しても構わないタイミング、例えば新しいトランザクションを開始した直後などに、メッセージを取得して、キャッシュの情報を削除する。ここで更新ではなく削除で十分な理由は、削除さえしておけば、次にその情報にアクセスする際に最新の情報が読み込まれるからである。キャッシュ情報の削除処理は、それぞれのバックエンドごとに異なるタイミング

で発生する。

4.1. データ構造

共有 inval メッセージを管理しているデータ構造は、図 4-2 ようになっている。全体を管理する構造体が SISeg 構造体であり、共有 inval メッセージ用のバッファ（buffer 配列）と、バックエンドプロセスごとの情報を管理するエントリ（procState 配列）を含んでいる。

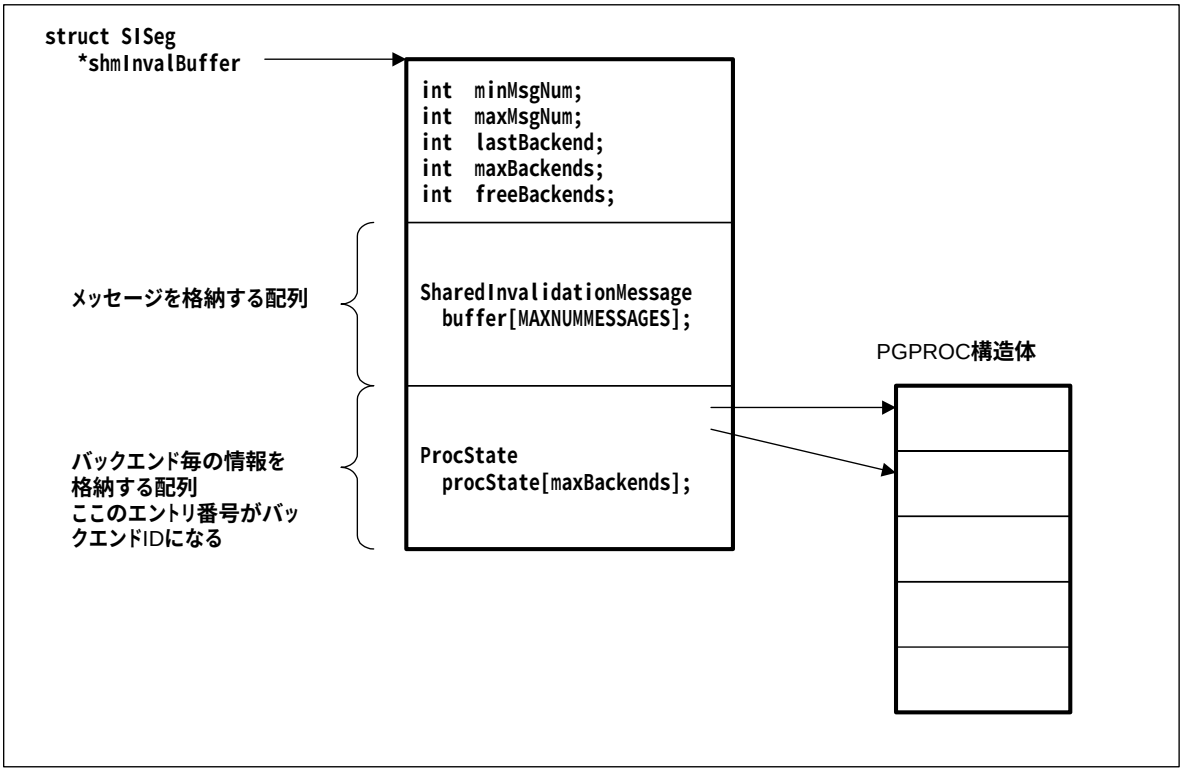

図 4-2 共有 inval メッセージを管理するデータ構造

4.1.1. procState 配列とバックエンド ID

図 4-2 の SISeg 構造体の procState 配列は、バックエンドごと共有 inval メッセージの使用状態を示している。実は、この配列はバックエンドごとの情報を管理するために非常に便利なため、バックエンドのプロセス情報の管理にも使用される。まず、この配列のインデックス番号＋１がバックエンドプロセス番号になっている。次に、この配列のエントリの中に、バックエンドプロセスごとの情報を管理するための PGPROC 構造体配列へのポインタ（共有メモリなので、厳密にははにはオフセット）が保存されている。

ProcState 構造体は、次のように定義されている。

```
/* 共有 inval 構造体の中に入る、バックエンドごとの情報 */
typedef struct ProcState
{
    /* ProcState 配列のエントリが、アクティブでない場合 nextMsgNum が -1 である */
    int         nextMsgNum;     /* 次に読み込むメッセージ番号か、または-1 */
    bool        resetState;     /* バックエンドがステートをリセットしなければなら
```

```
                                ない場合は true である */
    SHMEM_OFFSET procStruct;    /* バックエンドの PGPROC 構造体の位置 */
} ProcState;
```

nextMsgNum は、このバックエンドプロセスが次に読み出すメッセージ番号である。これは、次に読むメッセージがない場合、-1 である。読み出すメッセージがある場合、そのメッセージの番号を保持している。メッセージバッファが一杯のため、全てのキャッシュを捨てなければならない状態（後述）を示すフラグが resetState である。そして、PGPROC 構造体への共有メモリ中のオフセットを示すのが procStruct である。

以前は、PGPROC 構造体へのアクセスは、共有メモリ中のハッシュテーブルを介して行われていたらしい。しかし、モジュール構成は汚くなるが、高速化のため、この ProcState 配列を利用してアクセスするようにした経緯があるらしい。

SISeg 構造体に話を戻すが、バックエンドの情報を管理している ProcState 配列がいくつあるかを maxBackends で管理している。そして、空きエントリ数がいくつかあるかを freeBackends で管理している。これらを使って、接続できるバックエンドの上限を管理する。
そして、lastbackend であるが、ProcState 配列は先頭から詰めて使用するため、最後尾はどこかを示すのが、この変数である。これは、バックエンド全体を検査するときに for ループを使用するのだが、それを最短時間で終わらせるための配慮である。先頭から詰めて使っていても、バックエンドが終わる順番は一定ではないので、隙間ができるのが普通である。バックエンドが終了する際には、可能であれば lastbackend をできるだけ前まで詰める処理が入っている（図 4-3）。

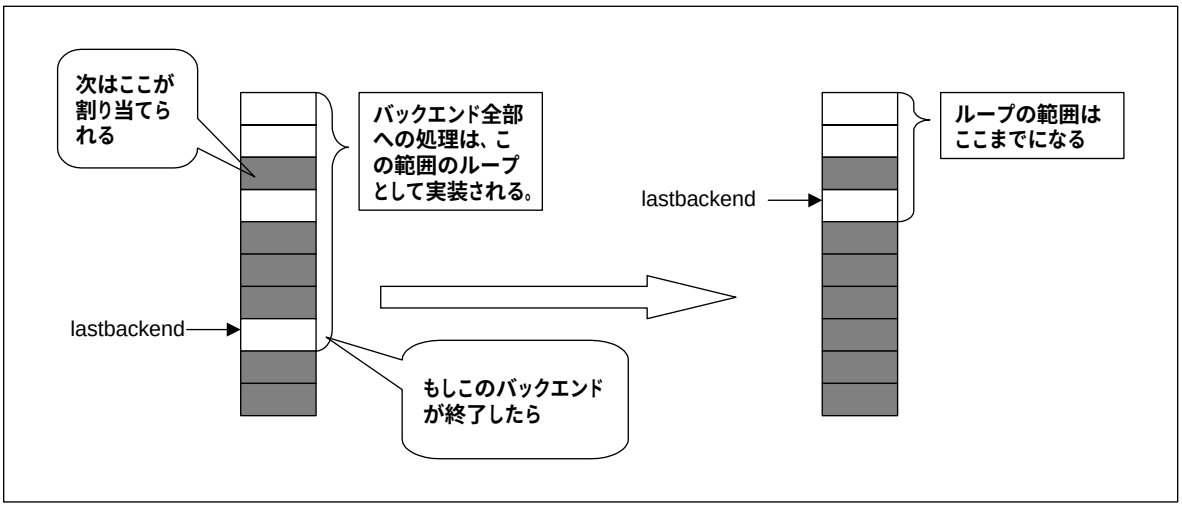

図 4-3 lastbackend による procState 配列の終端管理

4.1.2. キュー（共有 inval メッセージのバッファ）

キャッシュのオブジェクトが失効したことを伝えるメッセージのバッファ(図 4-2 の buffer 配列)は、固定で取られている。マクロ MAXNUMMESSAGES で定義されており、4096 に定義されている。つまり、システム全体で 4096 個の共有 inval メッセージを登録できる。

各バックエンドは、このバッファをキューとして使用する。

キューのイメージを示したのが次の図である。

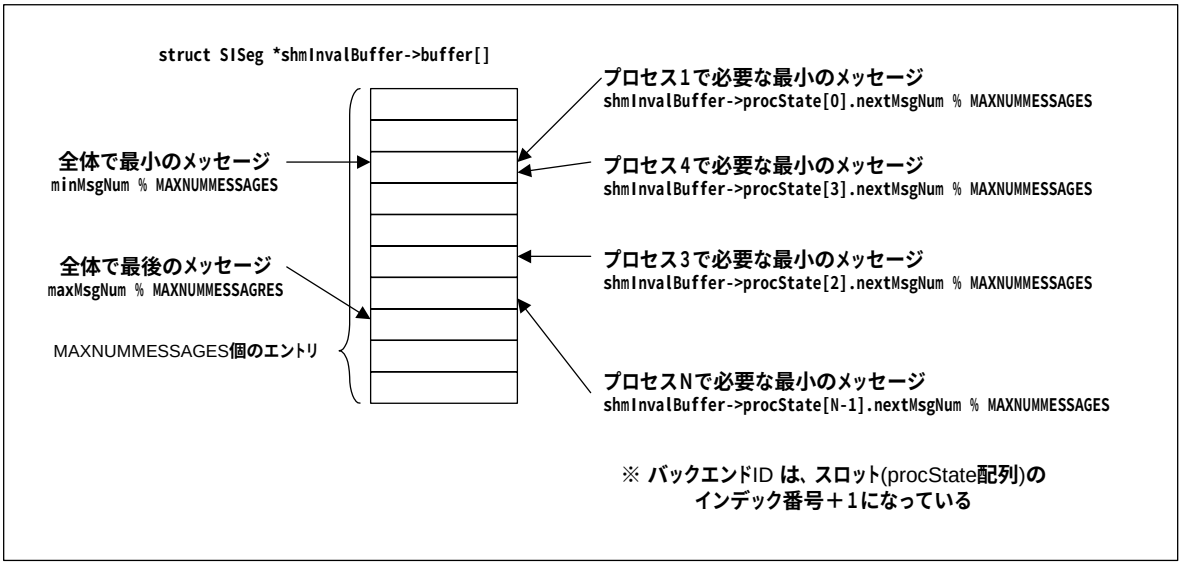

図 4-4 共有 inval メッセージのキュー

inval メッセージを格納するバッファ buffer は、リングバッファとして使われる。つまり、一番下まで来たら、先頭に戻って再利用される。リングバッファを使い切ったことを検出するため、全体で使用している最小のメッセージ番号と次に割り当てるべき最大メッセージ番号を管理している。もし、最大メッセージ番号が、最小メッセージ番号のためのバッファを追い越す場合、全てのプロセスの最小メッセージを管理している procState のリセットフラグを true にする。リセットフラグが設定された各バックエンドプロセスは、全てのキャッシュが無効になったという扱いになる。つまり、キャッシュの内容を捨てて、全て読み直すことになる。リセットは、実装によっては一定量ずつ行うことも可能だと考えられるが、現在の PostgreSQL の実装では、全てのバッファを一斉に無効にし、全てのバックエンドのキャッシュが一斉に無効になる。

リングバッファになっているので、全体で最小のメッセージのバッファのエントリは、図 4-4 のように、minMsgNum % MAXNUMMESSAGES で計算できる。同様に、他のメッセージもメッセージ番号に対して MAXNUMMESSAGES の剰余を計算することで、バッファの位置を計算できる。

メッセージ番号は単調増加することになるため、オーバーフローの対策も入っている。全体で最小のメッセージ番号が MSGNUMWRAPAROUND（標準では MAXNUMMESSAGES * 4096）を超えると、全てのメッセージ番号から MSGNUMWRAPAROUND を引くことで、メッセージ番号がオーバフローしないようになっている。これは、メッセージの削除処理 SIDelExpiredDataEntries()で行われる。

4.1.3. メッセージの種類とメッセージのデータ構造

メッセージには２つの種類がある。１つは、カタログキャッシュの失効を伝えるメッセージで、もう１つはリレーションキャッシュの失効を伝えるメッセージである。それぞれのメッセージは、違う構造になっているので、次のように union で定義されている。

```
typedef union
{
    int16       id;             /* type field --- must be first */
    SharedInvalCatcacheMsg cc;
    SharedInvalRelcacheMsg rc;
} SharedInvalidationMessage;
```

どちらのメッセージか識別するために先頭は 16bit の int である。union 中では、int16 も定義されているが、これが単独で使われることはない。
先頭の int16 の id であるが、0 以上の場合は、カタログキャッシュのメッセージであることを表す。これは、キャッシュ ID を兼ねている。-1 のときが、リレーションキャッシュのメッセージである。それ以外は、今後の拡張に利用できる。

カタログキャッシュ用のメッセージ構造体 SharedInvalCatcacheMsg は、次のように定義されている

```
typedef struct
{
    int16       id;             /* cache ID --- must be first */
    ItemPointerData tuplePtr;   /* tuple identifier in cached relation */
    Oid         dbId;           /* database ID, or 0 if a shared relation */
    uint32      hashValue;      /* hash value of key for this catcache */
} SharedInvalCatcacheMsg;
```

カタログキャッシュ用のメッセージには、カタログキャッシュのエントリを識別するために必要な情報が入っている。

リレーションキャッシュ用のメッセージ構造体 SharedInvalRelcacheMsg は、次のように定義されている。

```
typedef struct
{
    int16       id;             /* type field --- must be first */
    Oid         dbId;           /* database ID, or 0 if a shared relation */
    Oid         relId;          /* relation ID */
} SharedInvalRelcacheMsg;
```

リレーションキャッシュ用のメッセージは、どのデータベースのどのリレーションが失効したかを伝えられるだけの非常に単純なものである。

4.2. インターフェース

共有 inval メッセージのインタフェースは次のようになっている。

InitBackendSharedInvalidationState()	新しいバックエンドプロセスが、SISeg 構造体の procState 配列のエントリを確保する。つまりバックエンド ID が取得できる。
SendSharedInvalidMessage()	共有 inval メッセージを、共有 inval メッセージのキュー（リングバッファ）に追加する。例えば、あるカタログ情報のオブジェクトが更新された際に、キャッシュが無効になったことを伝える。
ReceiveSharedInvalidMessages()	キューに溜まっている sinval メッセージを全て処理する。内部にループがあり、すべての共有 inval メッセージを処理するまで実行されて、最新の状態になる。 ループの処理中に、SInvalLock を共有モードで取得するが、該当バックエンドのスロットにアクセスするのは、他にないという特性を利用して、該当スロットの更新を行っている。これによって、この関数のループの部分は、並列実行可能である。 全てのメッセージの処理が完了したら、全てのバックエンドの使用済みメッセージ削除処理を行う関数を呼び出す。このフェーズでは、排他ロックを取得するため、並列実行は行えない。

これらのインタフェースの使われ方は、次のようになっている。
バックエンドが生成されて認証などが完了したあと、InitBackendSharedInvalidationState()を呼ぶことで、バックエンド ID が取得できる。つまり、procState 配列のエントリを１つ確保する。
カタログやリレーションの更新処理の際に、SendSharedInvalidMessage()で更新したオブジェクトの情報を登録する。各バックエンドが、カタログなどを更新しても構わないタイミングで、ReceiveSharedInvalidMessages() を呼び出すことによって、そのバックエンドのキャッシュが最新の状態になる。ただし、最新と言っても、無効なものを捨てただけで新しいものをキャッシュするわけではない。

これらのインタフェースを実装するための関数は、src/backend/storage/ipc/sinvaladt.c に入っている。主な内部インタフェースとして、次のようなものがある。

SInvalShmemSize()	共有 inval メッセージの管理に必要な共有メモリのサイズを返す。
SIBufferInit()	共有メモリセグメント中に、共有 inval メッセージの領域を確保して、その中を初期化する（図 4-2 の SISeg 構造体の領域を確保して初期化する）。
SIBackendInit()	新しいバックエンドのために、SISeg 構造体の procState 配列のスロットを確保する。空きスロットがなかったら、最大同時接続数のバックエンドが接続されているということである。

	戻り値は、スロットが見つかったらスロット番号＋１の数字（バックエンドＩＤ)、空きスロットがなければ 0 、その他のエラーなら負の数を返す。
CleanupInvalidationState()	この関数を呼び出したバックエンドをアクティブでない状態にする。つまり、procState 配列のスロットを開放する。 この関数は、バックエンドのシャットダウン中に on_shmem_exit() から実行されるコールバック関数である。
SIInsertDataEntry()	新しい inval メッセージをキューに追加する。もし、バッファがフルでメッセージを追加できなかったら、バッファをクリアして、各バックエンドのリセットフラグを設定する。 また、バッファを 70%使っている時点で、バッファのオーバーフロー対策を行う（後述)。
SISetProcStateInvalid()	SIInsertDataEntry() の補助関数。バッファフルの場合に、バッファのクリアと各バックエンドのリセットフラグの設定を行う。
SIGetDataEntry()	引数で指定したバックエンド用の､次の inval メッセージを１つ取り出す。 戻り値は、最後のメッセージまで取り出し終わっていれば 0 、次のメッセージを取り出せれば 1 、リセットフラグが設定されていれば –1 を返す。
SIDelExpiredDataEntries()	全てのバックエンドが使用済みのメッセージの削除を行う。図 4-4 の「全体で最小のメッセージ」へのポインタを全てのバックエンドで最小のメッセージ番号まで進める。 また、この関数内で、必要に応じてメッセージ番号のオーバーフロー対策が行われる。

4.2.1. キャッシュ失効処理の関数

ReceiveSharedInvalidMessages()のときに呼び出される、キャッシュの失効処理を受け持つ関数を、もう少し詳しく見てみる。

実際にキャッシュを管理しているモジュールは、src/backend/utils/cache ディレクトリ以下にある。その中で、キャッシュの失効を取り扱っているのが inval.c である。キャッシュの失効処理を受け持つ関数もこの中で定義されている。

ReceiveSharedInvalidMessages() の呼び出しは、AcceptInvalidationMessages() の１ヶ所だけである。ReceiveSharedInvalidMessages() は、引数として関数へのポインタを受け取る。次のような引数で呼び出しているので、失効処理が LocalExecuteInvalidationMessage() 関数、リセット処理が InvalidateSystemCaches() 関数となる。

```
void AcceptInvalidationMessages(void)
{
    ReceiveSharedInvalidMessages(LocalExecuteInvalidationMessage, InvalidateSystemCaches);
}
```

LocalExecuteInvalidationMessage() 関数が呼ばれるのは、ReceiveSharedInvalidMessages() からであり、呼び出されるときに引数として、共有 inval メッセージを１つ受け取る。受け取った共有 inval メッセージの id が 0 以上、つまり、カタログキャッシュ用のメッセージの場合、その id のカタログキャッシュのエントリを削除し、その id のカタログキャッシュ用コールバック関数が登録されていれば、それを実行する。受け取った共有 inval メッセージがリレーションキャッシュ用のメッセージだったら、リレーションキャッシュからメッセージ中に記述されているリレーション ID のリレーション情報を削除し、リレーションキャッシュ用のコールバック関数が登録されていた場合、それを実行する。

InvalidateSystemCaches() は、そのバックエンドプロセスの ProcState のエントリにリセットフラグが設定された場合、つまり、共有 inval メッセージのバッファが一杯になってこのバックエンドが共有 inval メッセージによる修正ではキャッシュの修正が追いつかなくなった場合に実行される。処理は、非常に単純で、カタログキャッシュのリセット処理とリレーションキャッシュのリセット処理をそれぞれ実行し、登録されているカタログキャッシュ用のコールバック関数やリレーションキャッシュ用のコールバック関数があれば、それを実行する。カタログキャッシュ、リレーションが全てリセットされるので、キャッシュの不整合はなくなる。

4.2.2. キュー（バッファ）のオーバーフロー対策

SIInsertDataEntry() でメッセージを追加する際に、キューが一杯でないかチェックする。一杯になっているとキューをクリアして、全てのバックエンドのキャッシュを無効にしなければならないのだが、この処理は非常にコストのかかる処理になってしまうので、なるべくキャッシュのリセットは起こらないようにしている。
そのため、いくつかのキューのオーバーフロー対策が入っている。１つは、キューが一杯だったときに、リセットを行う前に SIDelExpiredDataEntries() を実行することで、空きエントリが作れないか試みることである。
もう１つは、長時間クライアントのリクエスト待ちになっているバックエンドが、共有 inval メッセージを処理していない可能性があるので、キューが 70%埋まった時点で、リクエスト待ちのバックエンドに共有 inval メッセージの処理を促すことである。

後者について、もう少し詳しく説明する。SIInsertDataEntry()でメッセージを追加したら、キューの使用率を調べ、70% 埋まっていたら WAKEN_CHILDREN をシグナルとして postmaster に送る。そうすると、postmaster は SIGUSR2 シグナル（NOTIFY メッセージ）を全てのバックエンドに送る。SIGUSR2 シグナルを受け取ったバックエンドの中で、アイドル状態だったものは、システムテーブル pg_listener を調べるトランザクションを実行する。このトランザクション開始の副作用として、アイドル状態のバックエンドは未処理の共有 inval メッセージを読み込む。これは、クエリを実行しているアクティブなバックエンドでは、決して実行されない。

4.3. PGPROC へのエントリポイントとしての共有 sinval メッセージモジュール

PGPROC 構造体配列は、バックエンドプロセスの情報を管理しているデータ構造である。PGPROC 構造体配列のインタフェースは、共有キャッシュの失効処理と直接関係ないので、sinval.c に入って

いるのは少し変である。しかし、性能上の理由から、PGPROC 構造体配列のインタフェースは、sinval.c モジュールに統合されている。
かつては、PGPROC 構造体配列のインタフェースは、ShmemIndex ハッシュテーブルを使うように実装されていたらしく、そのため性能が悪かったようである。
sinval セグメントの中にある procState 配列は、システム中で唯一、バックエンドごとのデータの配列の入っている場所であり、バックエンドの PGPROC 構造体へのポインタを保持するのに、最も便利な場所である。現在は、PGPROC 構造体配列のインタフェースは、sinval の ProcState 配列を使った簡単なループとして実装してあり、性能上の問題に対処している。

4.3.1. PGPROC 関連のインタフェース

PGPROC にアクセスするためのインタフェースとして、次の関数が定義されている。

DatabaseHasActiveBackends()	引数に与えられたデータベースを使って処理を実行しているバックエンドがあれば、true を返す。もし、引数 ignoreMyself が true の場合、自分のプロセスは無視する。 この関数は、DROP DATABASE と連動して使われ、アクティブなバックエンドが残っている間にデータベースを削除しないようにするために使われる。 注意：新しく開始するバックエンドが、削除しようとしているデータベースに接続しようとする可能性を、ここで検出することはできない。それで、バックエンドの開始処理側で注意を払う必要がある[9]。
TransactionIdIsInProgress()	引数に与えられたトランザクション ID が、どれかのバックエンドが実行中のトランザクション ID ならば、true を返す。
GetOldestXmin()	現在実行中の全てのトランザクションで、それらが実行を開始した時点で実行されていたトランザクションの中で、もっとも古いトランザクションの ID を返す。この関数は、VACUUM を実行したときに、テーブル内の削除済みタプルを保存しておかなければならないかどうかを決めるために使われる。 引数 allDbs が true の場合は、全てのバックエンドを対象として考える。引数 allDbs が false の場合、自分と同じデータベースで実行されているバックエンドを対象として考える allDbs = true は、共有リレーションのために必要である。そして、非共有リレーションは allDbs = false を使う。 注意: この関数で考慮する xid の集合の中に、現在実行中の全ての xid も含めている。これによって、もしちょうど開始したトランザクションで、まだ、スナップショットを設定していない

[9] 「postgresプロセスの概要」の資料を参照。バックエンド起動時に、何度かチェックを行い、最終的にデータベース名をロックすることで接続を確認する。

	ものがあったとしても、そのトランザクションがスナップショットを設定しようとしたとき、そのトランザクションは、ここで計算したxmin より小さい値を設定することはない。なぜなら、そのスナップショットは、現在実行中のxid相当か、そのいくつかの実行が終わったものになるはずだからである[10]。
GetSnapshotData()	実行中のトランザクションの情報をスナップショットとして返す（詳細は後述）
CountActiveBackends()	自分以外のバックエンドで、トランザクションがアクティブなバックエンドの数を返す。この関数は、commit の処理中に、XLOG flush delay の価値があるかどうかを決めるために使われる。
GetUndoRecPtr()	実行中のトランザクションの中で、最も古いログレコードへのポインタ PGPROC->logRec を返す。
BackendIdGetProc()	引数で与えられた BackendId の PGPROC 構造体を見つけて、そのポインタを返す。
CountEmptyBackendSlots()	バックエンドプロセステーブル (procState 配列) の空きエントリの数を返す。

この中で、スナップショット作成に使用する GetSnapshotData() 関数は、MVCC などの機能を理解するために必要なため、もう少し詳しく見ておく。

4.3.2. スナップショットの作成 GetSnapshotData()

PostgreSQLは追記型のストレージを持っており、あるタプルの全ての更新が履歴として保存されている[11]。あるタプルが有効であるかどうかは、そのタプルを生成したトランザクションのIDと削除したトランザクションのIDを管理することで実現している。

PostgreSQL では、ある瞬間に見ることのできるタプルの集合を保存しておくために、スナップショットを利用する。PostgreSQL のスナップショットの実装は、その瞬間に動いていたプロセスの全てのトランザクション ID を保存することである。

スナップショットは、次のようなデータ構造をしている。

```
typedef struct SnapshotData
{
    TransactionId xmin;         /* XID < xmin are visible to me */
    TransactionId xmax;         /* XID >= xmax are invisible to me */
    uint32      xcnt;           /* # of xact ids in xip[] */
    TransactionId *xip;         /* array of xact IDs in progress */
    /* note: all ids in xip[] satisfy xmin <= xip[i] < xmax */
    CommandId   curcid;         /* in my xact, CID < curcid are visible */
    ItemPointerData tid;        /* required for Dirty snapshot -:( */
} SnapshotData;
```

[10] ソースコードのコメントから引用している。PostgreSQLの場合、スナップショットを作成してからxidを取得する場合と、xidを取得してからスナップショットを作成する場合がある。後者の場合の対応となっているが、実際は、現在実行中のトランザクションの開始時点の最小xidが取得されるので、無駄な処理だと言える。

[11] 厳密には、全く使う必要がなくなったタプルは、VACUUM処理で回収される。

スナップショットには、最小のトランザクション ID **xmin** と最大のトランザクション ID **xmax** を保持している。これにより、xmin より小さいトランザクション ID や xmax より大きいトランザクション ID とスナップショットに含まれる各トランザクション ID との比較する場合、スナップショットの配列を調べる必要がなくなるので、計算のコストを減らすことができる。
xcnt には、いくつのトランザクション ID を **xip** 配列に格納してあるかを示している。xip 配列は、スナップショット作成時に実行されていたトランザクションのトランザクション ID を格納してる配列である。
このスナップショットを実行したコマンド ID を **curcid** とし保存しておくことにより、同じトランザクション中でも、このスナップショットよりあとの更新は見えないようにする。
tid については、今回は調べていない。

本資料の範囲である共有 inval メッセージモジュールには、スナップショットの使い方は含まれない。従って、スナップショットは、スナップショット作成時に実行されていたトランザクション ID を保持する機能であるという説明にとどめる。スナップショットとそれによる MVCC の実現については、次の資料としてまとめることにする。

＜EOF＞