rev 06.03.31

PostgreSQL 解析資料

～ バックグラウンドライタプロセス ～

(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、バックグラウンドライタプロセスについて説明する。バックグラウンドライタプロセスは、定期的にダーティバッファを書き出すことを主な目的として、PostgreSQL 8 .0 で実装されたサーバープロセスである。以下に、バックグラウンドライタプロセスで行われる仕事を挙げる。

◇ チェックポイント時に必要なダーティページの書き出し量の削減を目的とした、定期的なダーティページの書き出し（GUC bgwriter_all_percent / bgwriter_all_maxpages）
◇ バッファページの再利用に伴うダーティページの書き出し頻度の低減を目的とした、定期的なダーティページの書き出し（GUC bgwriter_lru_percent / bgwriter_lru_maxpages）
◇ 定期的なチェックポイントの実行（GUC checkpoint_timeout）
◇ 各バックエンドプロセスから要求されたチェックポイントの実行
◇ シャットダウン時におけるトランザクションログの終了処理の実行

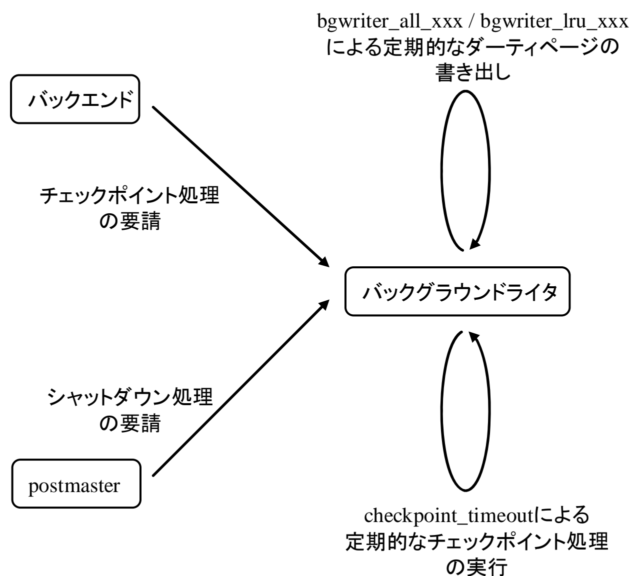

図 1 バックグラウンドライタの主な処理

Copyright © 2006 NTTDATA

1.1. 対象バージョン

本資料は、PostgreSQL 8.1.3 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。また、Windows のソースコード部については考慮していない。

2. 定期的なダーティページの書き出し処理

この章では、バックグラウンドライタプロセスの主な処理の 1 つである、定期的なダーティページの書き出しについて説明する。

バックグラウンドライタプロセスでは、メインループ内において、シャットダウンおよびチェックポイント処理が発生していなければ、GUC bgwriter_all_xxx と bgwriter_lru_xxx による定期的なダーティページの書き出し処理を実施する。書き出し処理は src/backend/storage/buffer/bufmgr.c の BgBufferSync()を定期的に呼び出すことになり、bgwriter_all_xxx / bgwriter_lru_xxx が持つ、それぞれのページ開始位置よりダーティページの書き出しを試みる。

2.1. GUC bgwriter_all_xxx によるダーティページの書き出し

bgwriter_all_xxx による定期的なダーティページの書き出し処理は、チェックポイント時に必要なダーティページの書き出し量の削減を目的としている。
ダーティページの書き出し開始位置を保持する変数は static 変数で宣言されており、バックグラウンドライタが起動された時点では 0 の位置である。BgBufferSync()で 1 回分のダーティページの定期書き出しが終了し、再度次の BgBufferSync()の呼び出しが行われると、前回の終了位置から引き続いて、次のページ位置から開始される。つまり、bgwriter_all_xxx による書き出し処理は、単純に共有バッファを順に周回して、見つかったダーティバッファを書き込む処理となっている。bgwriter_all_xxx によるダーティページの書き出し手順は、次のようになっている。

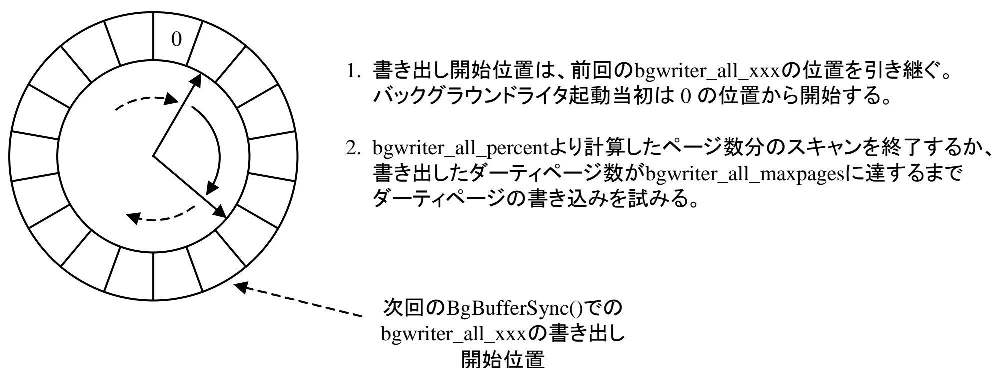

図 2 bgwriter_all_xxx によるダーティバッファの書き出し

1. GUC bgwriter_all_percent と bgwriter_all_maxpages の値をチェックする。
   どちらかの値が 0 以下の場合は、bgwriter_all_xxx によるダーティページの書き出し処理を行わない。
2. ダーティページを書き出した数を数えるカウンタ、num_written を 0 に初期化する。

Copyright © 2006 NTTDATA

3. bgwriter_all_percent より計算されたページ数分、共有バッファ内のページをスキャンするループを開始する。
4. 現在スキャン中のページ位置 buf_id1 が NBuffers-1 を超えたら、buf_id1 を 0 に戻す。超えていない場合は buf_id1 をインクリメントする。
5. SyncOneBuffer()を呼び出して、buf_id1 の指すページの書き出しを試みる。このページがダーティページであり、正常に書き出しが行われた場合には、書き出し済みカウンタ num_written をインクリメントする。num_written が bgwriter_all_maxpages を超えた場合はループを抜け、処理を終了する。
6. ループの先頭に戻る。

2.2. GUC bgwriter_lru_xxx によるダーティページの書き出し

バックエンドプロセスでのバッファページを再利用する場合、割り当てられたバッファがダーティならば、バッファを書き出してから利用する。bgwriter_lru_xxx による定期的なダーティページの書き出し処理は、このバッファ割り当て時のダーティページの書き出し頻度の低減を目的としている。バッファの書き出し開始位置は、src/backend/storage/buffer/freelist.cのStrategySyncStart()で求められる。この位置は、clock sweepアルゴリズム[1]が次に走査する予定のページを指している。bgwriter_all_xxxによる書き出しでは、前回走査した最後の位置を引き継いでいたのに対して、bgwriter_lru_xxxによる書き出しでは、毎回StrategySyncStart()を呼び出して、その時点の書き出し位置を再計算する。bgwriter_lru_xxxによるダーティページの書き出し手順は、次のようになっている。

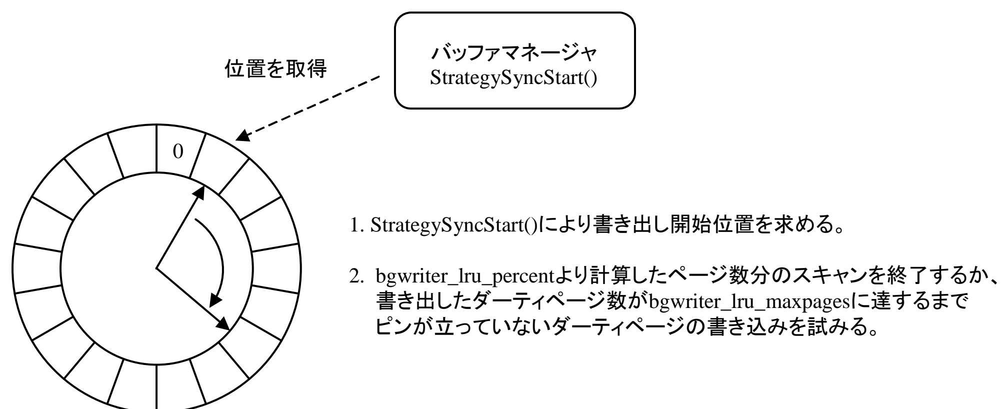

図 3 bgwriter_lru_xxx によるダーティバッファの書き出し

1. GUC bgwriter_lru_percent と bgwriter_lru_maxpages の値をチェックする。
   どちらかの値が 0 以下の場合は、bgwriter_lru_xxx によるダーティページの書き出し処理を行わない。
2. ダーティページを書き出した数を数えるカウンタ、num_written を 0 に初期化する。
3. StrategySyncStart()により書き出し開始位置を求め、buf_id2 を初期化。
4. bgwriter_lru_percent より計算したページ数分、共有バッファ内のページをスキャンするループを開始する。

[1] PostgreSQL8.1 から採用されているバッファ管理アルゴリズム。

Copyright © 2006 NTTDATA

5. 現在スキャン中のページ位置 buf_id2 が NBuffers-1 を超えたら、buf_id2 を 0 に戻す。超えていない場合は buf_id2 をインクリメントする。
6. SyncOneBuffer()を呼び出して、buf_id2 の指すページの書き出しを試みる。このページが、PIN が立っていないダーティページであり、正常に書き出しが行われた場合には、書き出し済みカウンタ num_written をインクリメントする。num_written が bgwriter_lru_maxpages を超えた場合はループを抜け、処理を終了する。
7. ループの先頭に戻る。

3. チェックポイントの契機

この章では、チェックポイントの要求処理と、その他のチェックポイントの契機について説明する。

各バックエンドプロセスでは、バックグラウンドライタプロセスに対してチェックポイントシグナル(SIGINT)を発行することで、バックグラウンドライタにチェックポイント処理を要求する。シャットダウン時におけるチェックポイント処理の実行は、postmaster からバックグラウンドライタプロセスに対してシャットダウンシグナル(SIGUSR2)を発行し、バックグラウンドライタ側のシャットダウン処理でチェックポイントが実行される。

バックグラウンドライタプロセスに対するチェックポイントシグナルの発行は、src/backend/postmaster/bgwriter.cのRequestCheckpoint()を呼び出すことによって行われる。関数仕様は、表 1のようになっている。

表 1 RequestCheckpoint()の関数仕様

関数	void RequestCheckpoint(bool waitforit, bool warnontime)
引数	bool waitforit ： true の場合、チェックポイントが完了するまで待つ。 bool warnontime ： true の場合、GUC の checkpoint_warning による警告メッセージの出力を有効にする。
戻り値	なし。
処理内容	バックグラウンドライタとバックエンド間のコミュニケーション用の共有メモリ領域 BgWriterShmem に値を設定した後、バックグラウンドライタプロセスに対して SIGINT を発行する。その後、waitforit が true の場合は BgWriterShmem の内容をチェックして、チェックポイント処理が完了していなければ一定時間スリープし、再度 BgWriterShmem の内容のチェックを繰り返してチェックポイント処理が完了するまで待つ。

RequestCheckpoint()を呼び出すチェックポイントの要求契機とその引数値は、表 2のとおりである。シグナルを受け取ったバックグラウンドライタ側ではチェックポイント処理を実行する[2]。

[2] ただし、スタンドアロンモード時においては、RequestCheckpoint() 内で直接チェックポイント処理を呼び出している。

Copyright © 2006 NTTDATA

表 2 RequestCheckpoint()によるチェックポイントの要求契機

RequestCheckpoint()の要求契機	引数 waitforit	引数 warnontime
VACUUM 処理での CLOG の切り詰め処理時	true	false
オンラインバックアップ開始時	true	false
CREATE DATABASE コマンド時	true	false
DROP DATABASE コマンド時（WIN32 のみ）	true	false
CHECKPOINT コマンド時	true	false
トランザクションログセグメント数が GUC checkpoint_segments に達した場合	false	true

RequestCheckpoint()を介さずに、直接CreateCheckPoint()を呼び出してチェックポイントを実行する契機は、表 3のとおりである。

表 3 RequestCheckpoint()を介さないチェックポイントの実行契機

チェックポイント処理の実行契機
initdb 時の BootstrapMain()の最後
サーバ起動時のリカバリ処理の完了時（スタートアッププロセス）
シャットダウン時（バックグラウンドライタプロセス）
前回のチェックポイントから GUC checkpoint_timeout 秒を経過した場合（バックグラウンドライタプロセス）

4. バックグラウンドライタプロセスの起動

postmasterの起動処理において、スタートアッププロセスが終了した後にpostmasterの子プロセスとして、バックグラウンドライタプロセスは起動される[3]。バックグラウンドライタのプロセスIDはpostmasterで管理されており、もしプロセスがいないことを検出したら、プロセスの再起動を試みる。ソースコード上でのプロセスの起動は、マクロ StartBackgroundWriter()を呼び出すところから始まる。マクロ StartBackgroundWriter()は次のように定義されている。

```
#define StartBackgroundWriter() StartChildProcess(BS_XLOG_BGWRITER)
```

StartChildProcess()では fork()で子プロセスを起動して、BootstrapMain()関数を呼び出す。そして定数 BS_XLOG_BGWRITER により、バックグラウンドライタプロセスのメインエントリポイントとなる src/backend/postmaster/bgwriter.c の BackgroundWriterMain()関数を呼び出すことになっている。

5. バックグラウンドライタプロセスの終了

postmaster の終了処理において、バックグラウンドライタプロセスに対してシャットダウンシグナル

[3] 統計情報収集、auto vacuum、WALアーカイバ、sysloggerプロセスなども、postmasterから子プロセスとして起動される。

Copyright © 2006 NTTDATA

(SIGUSR2)を発行することにより、バックグラウンドライタプロセスでは終了処理を開始する。
終了処理では、ShutdownXLOG()を呼び出してトランザクションログの終了処理を行った後に、proc_exit()でプロセスを終了する。即時終了シグナル(SIGQUIT)を検出した場合の終了処理については、exit()ですぐにプロセスを終了する。

6. バックエンド～バックグラウンドライタ間のコミュニケーション用領域

postmaster の起動の際、共有メモリ上には、バックエンドとバックグラウンドライタ間のコミュニケーション用領域が確保される。コミュニケーション用領域 BgWriterShmemStruct 構造体は、次のように定義されている。

```
typedef struct
{
    RelFileNode rnode;
    BlockNumber segno;
    /* might add a request-type field later */
} BgWriterRequest;

typedef struct
{
    pid_t       bgwriter_pid;   /* PID of bgwriter (0 if not started) */

    sig_atomic_t ckpt_started;  /* advances when checkpoint starts */
    sig_atomic_t ckpt_done;     /* advances when checkpoint done */
    sig_atomic_t ckpt_failed;   /* advances when checkpoint fails */

    sig_atomic_t ckpt_time_warn;    /* warn if too soon since last ckpt? */

    int         num_requests;   /* current # of requests */
    int         max_requests;   /* allocated array size */
    BgWriterRequest requests[1];    /* VARIABLE LENGTH ARRAY */
} BgWriterShmemStruct;

static BgWriterShmemStruct *BgWriterShmem;
```

バックグラウンドライタ内では、チェックポイント処理 CreateCheckPoint()を呼び出す前に ckpt_started メンバをインクリメントし、チェックポイント処理が完了した後に、ckpt_started メンバの値を ckpt_done メンバに格納している。これらの値を比較することで、RequestCheckpoint()では、バックグラウンドライタ側で実行されるチェックポイント処理の終了待ち判定を行っている。

ckpt_failed メンバは、バックグラウンドライタのエラー発生時にインクリメントされる。RequestCheckpoint()では、この値を元にチェックポイントのエラー判定を行っている。

ckpt_time_warn メンバは警告メッセージの出力を有効にするかどうかのフラグとして使用される。RequestCheckpoint()では、引数 warnontime の値を ckpt_time_warn メンバにセットした上で、チェックポイントを要求している。バックグラウンドライタ側では、GUC checkpoint_warning による警告メッセージの出力判定の際に参照される。

Copyright © 2006 NTTDATA

num_requests、max_requests、requests メンバは、ストレージマネージャ smgr の fsync リクエストキューの更新ために使用されている。

7. バックグラウンドライタにおけるシグナルハンドラの定義

バックグラウンドライタプロセスで定義されているシグナルハンドラは、表 4のようになる。
基本的にシグナルハンドラでは、各シグナルの検出フラグをセットするだけであり、シグナルに従った実処理は、バックグラウンドライタプロセスのメインループ内で、各シグナルの検出フラグ判定を行った上で処理することになっている。

表 4 シグナルハンドラ

シグナル	シグナルハンドラ	処理内容
SIGHUP	BgSigHupHandler()	設定ファイルの読み直しを行う。シグナルハンドラでは、got_SIGHUP フラグをセットするのみ。
SIGINT	ReqCheckpointHandler()	チェックポイントを実行する。シグナルハンドラでは、checkpoint_requested フラグをセットするのみ。
SIGQUIT	bg_quickdie()	即時終了を行う。シグナルハンドラ内で exit(1)を実行して、プロセスを終了する。
SIGUSR2	ReqShutdownHandler()	シャットダウン処理を行った後、プロセスを終了する。シグナルハンドラでは、shutdown_requested フラグをセットするのみ。

上記以外のシグナルには SIG_IGN を設定して、シグナルを無視している。

8. BackgroundWriterMain()の処理シーケンス

この章では、バックグラウンドライタプロセスが起動された後の処理の流れについて説明する。

バックグラウンドライタプロセスのメインエントリポイント、BackgroundWriterMain()の処理シーケンスは次のようになっている。

1. シグナルハンドラの設定。
2. エラー発生時の処理。sigsetjmp()により呼び出される。
3. メインループの開始。
4. チェックポイントに関するフラグのクリア。チェックポイント実行フラグ(do_checkpoint)と、強制チェックポイントフラグ(force_checkpoint)を false にする。
5. AbsorbFsyncRequests()を呼び出して smgr の fsync リクエストキューを更新する。
6. 設定ファイルの読み直しシグナルの検出判定。
   got_SIGHUP フラグが true ならば、ProcessConfigFile()で設定ファイルを読み直す。
7. チェックポイントシグナルの検出判定。

Copyright © 2006 NTTDATA

checkpoint_requested フラグが true ならば、チェックポイント実行フラグ(do_checkpoint)と、強制チェックポイントフラグ(force_checkpoint)を true にする。

8. シャットダウンシグナルの検出判定。
   shutdown_requested フラグが true ならばシャットダウン処理を行い、proc_exit(0)でプロセスを終了する。なお、ここで呼び出されるトランザクションログの終了処理 ShutdownXLOG()において、シャットダウンチェックポイントが実行される。
9. GUC checkpoint_timeout の時間経過判定。
   前回のチェックポイントから GUC checkpoint_timeout 以上の時間が経過していたら、チェックポイント実行フラグ(do_checkpoint)を true にする。この場合の強制チェックポイントフラグ(force_checkpoint)には何も値をセットしない。
10. チェックポイントの実行フラグ(do_checkpoint)の判定。
    ・do_checkpoint が true の場合

    最初にGUC checkpoint_warningによるメッセージ出力判定が行われる。これは、コミュニケーション用領域のckpt_time_warnがtrue[4]で、かつ前回のチェックポイントの実行からcheckpoint_warning秒より短い間隔でチェックポイントが要求された場合にのみ、GUC checkpoint_segments値の見直しを勧告するメッセージを出力する。

    次に、コミュニケーション用領域のckpt_startedをインクリメントした後、チェックポイント処理CreateCheckPoint()を実行する。CreateCheckPoint()の引数shutdownは常にfalseで、引数forceは強制チェックポイントフラグ(force_checkpoint)の値で呼び出す[5]。CreateCheckPoint()の実行後は、チェックポイント処理によるファイルの削除を確実にするために、smgrcloseall()を実行する。

    最後に、バックエンド側にチェックポイントの終了を伝えるために、コミュニケーション用領域の ckpt_started の値を ckpt_done に代入する。そして、GUC checkpoint_timeout と checkpoint_warning の判定材料となるチェックポイントの実行時間を更新する。

    ・do_checkpoint が false の場合

    GUC bgwriter_all_xxx / bgwriter_lru_xxx による、定期的なダーティページの書き出し処理、BgBufferSync() を実行する
11. GUC bgwriter_delay 値のスリープ。
12. メインループの先頭に戻る。

---

[4] 現状では、GUC checkpoint_segmentsによるチェックポイントのみとなる。
[5] 現状では、引数forceがfalseになる場合は、GUC checkpoint_timeoutによるチェックポイントのみとなる。

Copyright © 2006 NTTDATA

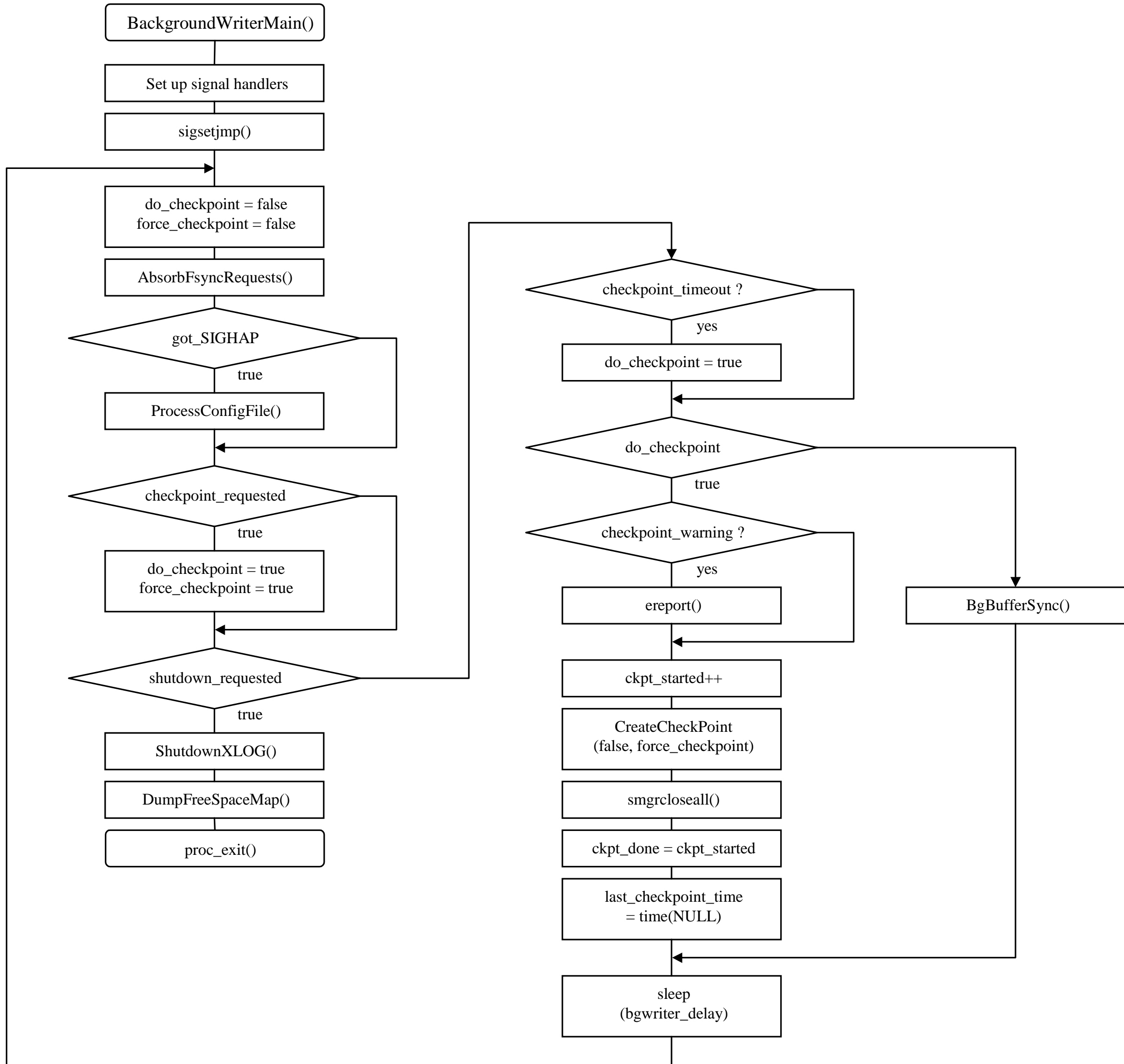

図 4 BackgroundWriterMain()の処理シーケンス

＜ＥＯＦ＞

Copyright © 2006 NTTDATA