2004.4.20 修正版

JPUG しくみ分科会 第4回勉強会

PostgreSQLソースコード解析

～ postmaster ～

2004年4月19日

NTTデータ先端技術(株)
井久保 寛明

Agenda

- Unix の基礎
  - シグナルハンドリング
  - プロセス間通信
- postmaster の概要
  - postmaster の仕事
- postmaster の実装
  - postmaster の設計思想
  - 起動シーケンス
  - サーバループ
  - クライアントの接続からバックエンドへ
  - チェックポイントの実行
  - DBのシャットダウン
  - BootstrapMain()への移行

対象は、基本的に7.4.2のpostmasterである
一部、古いバージョンについての説明あり

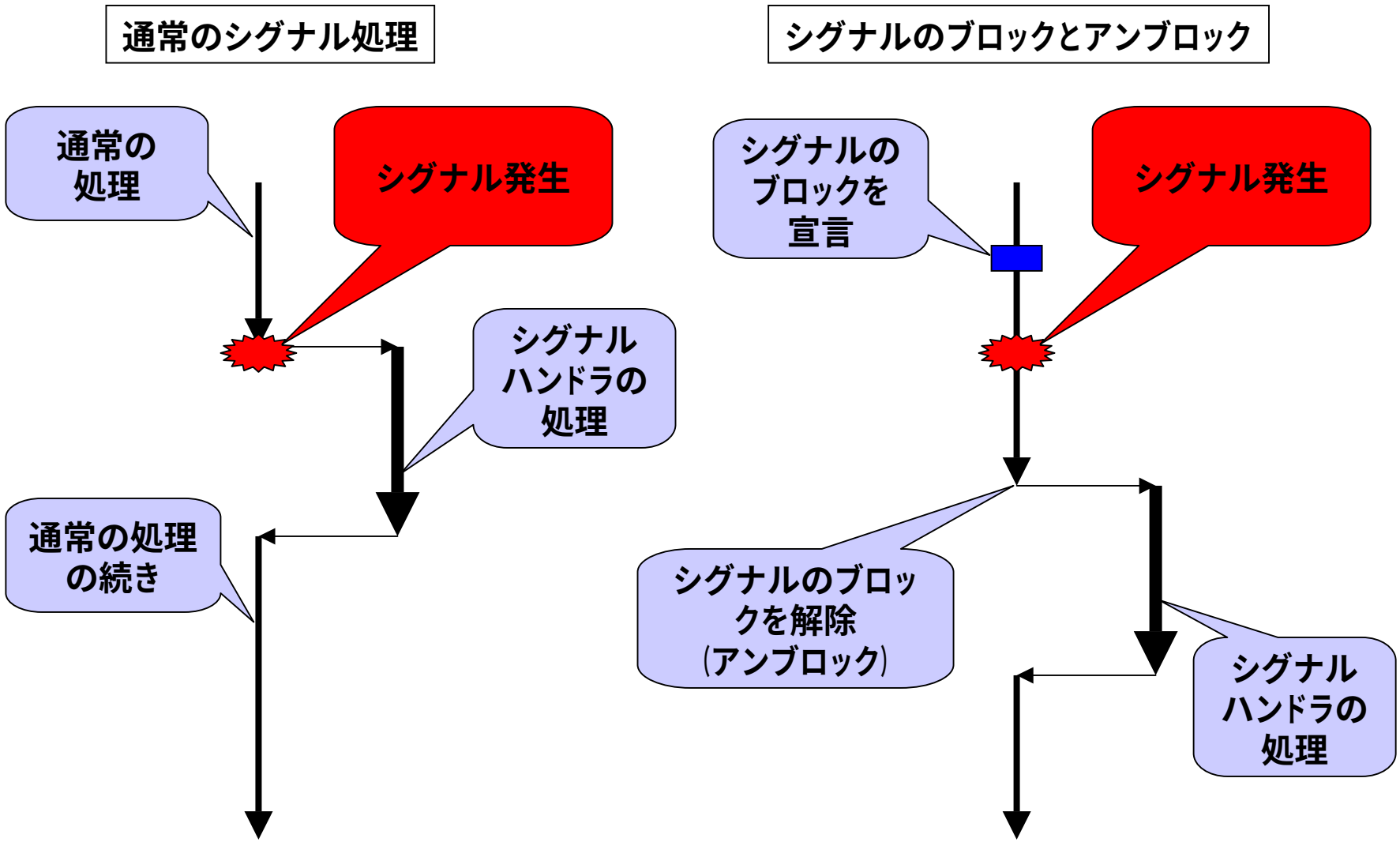
Unixの基礎

シグナル

■ シグナルとは？

- プロセスに対して、Unix カーネルが処理の割込みを通知をする手段
- 通知を受けたプロセスは、受け取ったシグナルの種類にあわせたシグナルハンドラを起動する
- シグナルハンドラの処理が終わったプロセスは、通常の処理に戻る
- 各シグナルに対しては、デフォルトの動作(シグナルハンドラ)が定義してあり、一部のシグナル(SIGKILL, SIGSTOP)を除いて、それらをユーザが定義したシグナルハンドラに置き換えることができる
  - シグナルハンドラのデフォルトの動作は、無視、停止、再開、abort、exitなどである
  - 無視は、そのシグナルが来ても何もしない
  - 停止はそのプロセスを一時停止させ、再開のシグナルが来たら処理を再開する
  - abort は、core と呼ばれるファイルにプロセスアドレス空間の内容とレジスタの内容を書き出して終了する (core dump)
  - exit は、 core dump を発生せずにプロセスを終了する
- シグナルを受け取りたくないとき(クリティカルセクションなどの処理中)は、シグナルをブロックして、あとで受け取ることができる

シグナル



シグナル

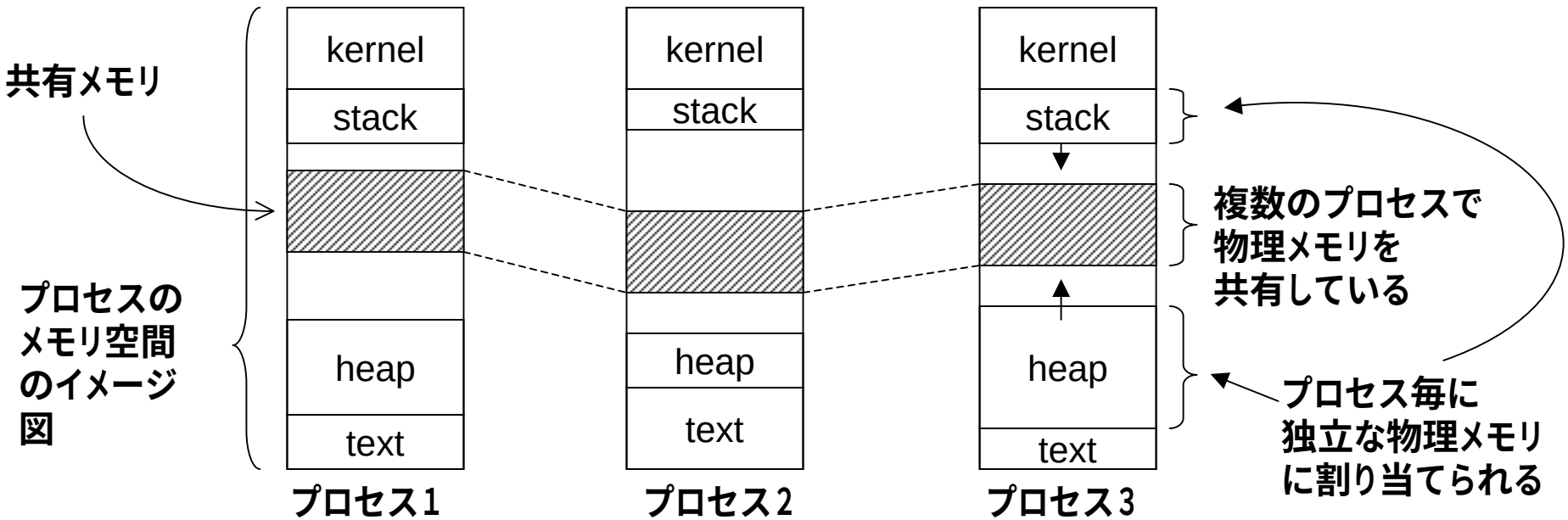
■ PostgreSQL に関連する主なシグナル

シグナル	デフォルトのシグナルの意味	PostgreSQLで使われ方
AIGALRM	時間アラーム	時間のアラームとして使われる
SIGCHLD	子プロセスが終了するかサスペンドされた	バックエンドや他の子プロセスの終了の通知。これを利用して、バックエンドの管理情報の削除などを行う
SIGHUP	ハングアップ	コンフィグファイルなどの読み直し
SIGTERM	プロセスの終了	実行中の全てのトランザクションの終了を待ってpostmasterを終了する
SIGINT	tty割込み (Ctrl – C)	実行中の全てのトランザクションをアボートして終了する
SIGQUIT	tty quit シグナル (Ctrl - ¥)	ログのフラッシュ程度で、即時終了を行う。
SIGUSR1	ユーザ定義用	共有メモリにフラグをセットしたことを、バックエンドから postmaster に通知するために使用
SIGUSR2	ユーザ定義用	バックエンドがpostmaster からの終了を受け取る

プロセス間通信（共有メモリ）

■ 共有メモリ

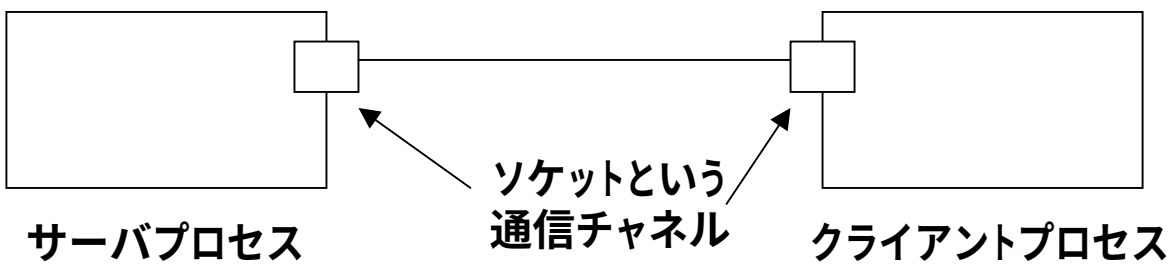
- ◆ 通常、各プロセスは他のプロセスのメモリ空間を見ることはできない。一部のメモリ空間を同一の物理メモリに割り当てて、複数のプロセスでメモリを共有するしくみである。
- ◆ 共有しているメモリの内容変更すると、他のプロセスでもすぐ反映されるので、通常は、排他制御のしくみと合わせて利用する。



プロセス間通信（ソケット通信）

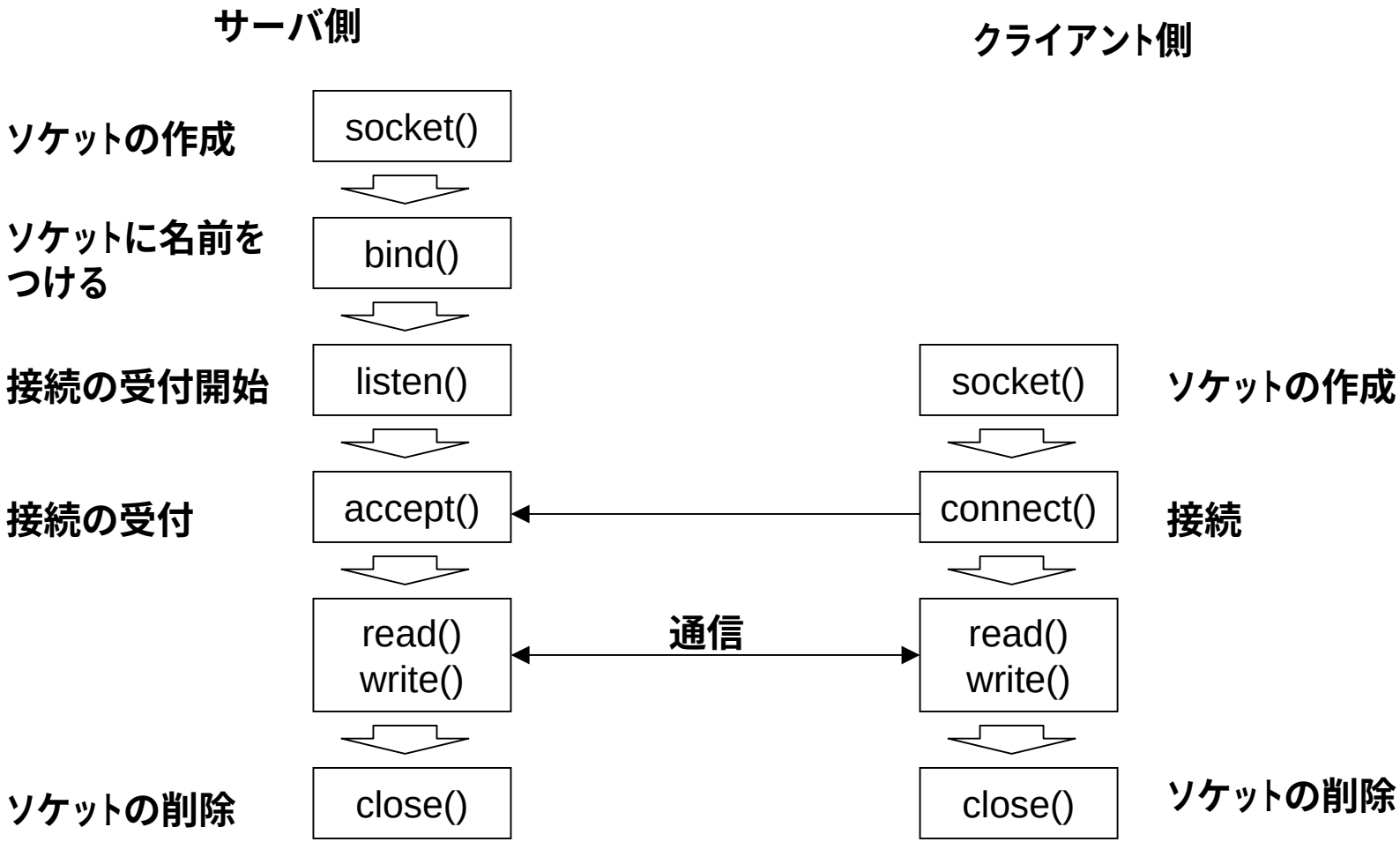
■ ソケット通信

- ◆ ソケットとは、Unixで使うことのできるプロセス間通信の機構の1つ
- ◆ ソケットのファイルデスクリプタを作って、read/write で読み書きができる
- ◆ コネクション型とコネクションレス型のソケットがあるが、PostgreSQLの通信で使うのは、コネクション型
- ◆ 通信相手の指定方法の定義域をドメインと呼ぶ
  - UNIX ドメイン
    - ▸ 1台のマシン上で一意であることを保証するドメイン
    - ▸ ファイル名を利用して、相手を指定する
  - INET ドメイン
    - ▸ IPアドレスとポート番号を指定して、相手を指定するドメイン

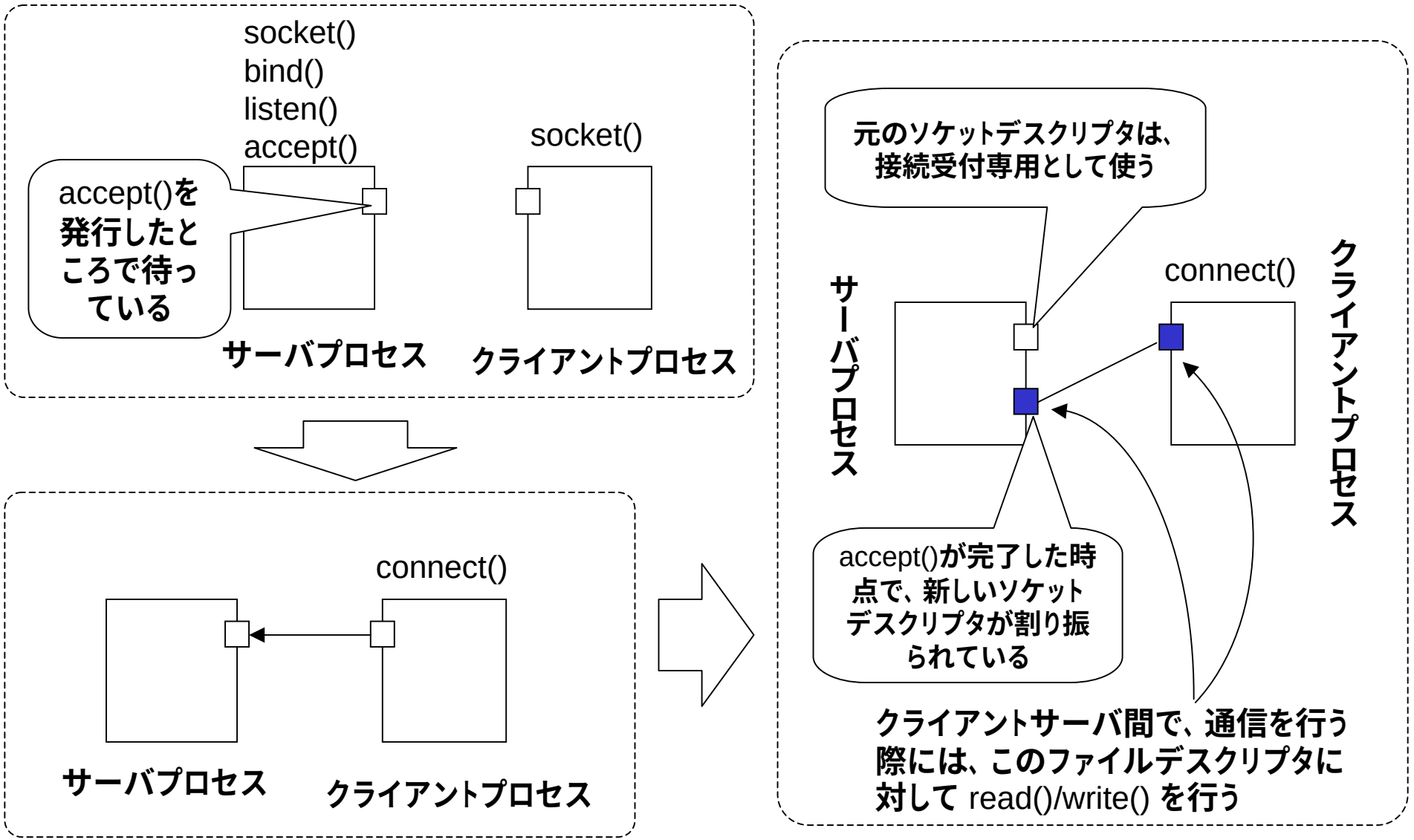


プロセス間通信（ソケットプログラミング）

■ ソケットプログラミング

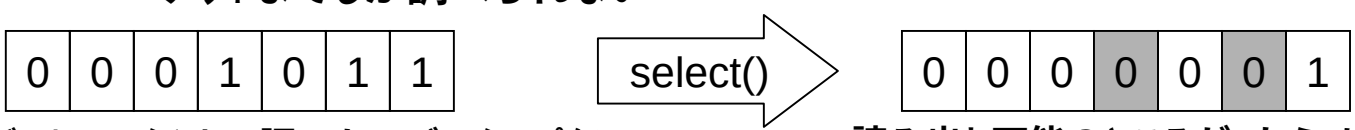


プロセス間通信（accept と ソケットデスクリプタ）



プロセス間通信（select システムコール）

- 複数のソケットに対して、読出し可能か、書込み可能か、エラーが発生しているを調べることができる
- postmaster では、UNIXドメインのソケットと複数のINETドメインのソケット(TCP/IP)のどれかに接続要求が来たことを検出するため、読出し可能ソケットを調べるために使用する
- select システムコールでは、タイムアウトが指定できる (struct timeval *)
  - 0 秒にすることで、待ち時間を0にすることもできる
  - また、ポインタをNULL にすることで、無限に待ち続けるようにすることもできる
- postmaster では、このタイムアウトを利用して、定期チェックポイントの時間待ちにも使用している
  - なんからの接続があってタイムアウトまでいかなかった場合、残り時間を再計算する
- select システムコールは、ビットマスクで調べたいソケットを調べる
  - ビットマスクの上限が1024に定義されていることが多いので、最大で同時に1024個のソケットまでしか調べられない



ビットマスクとして調べたいデスクリプタに1をセットする
普通は、FD_ZERO() と FD_SET() マクロをしようして設定する

読み出し可能のところが1から0に変わるので、ビットマスクとXORをとると、どのデスクリプタか判る。
普通は、FD_ISSET() マクロを使用して調べる

postmaster の概要

postmaster の仕事

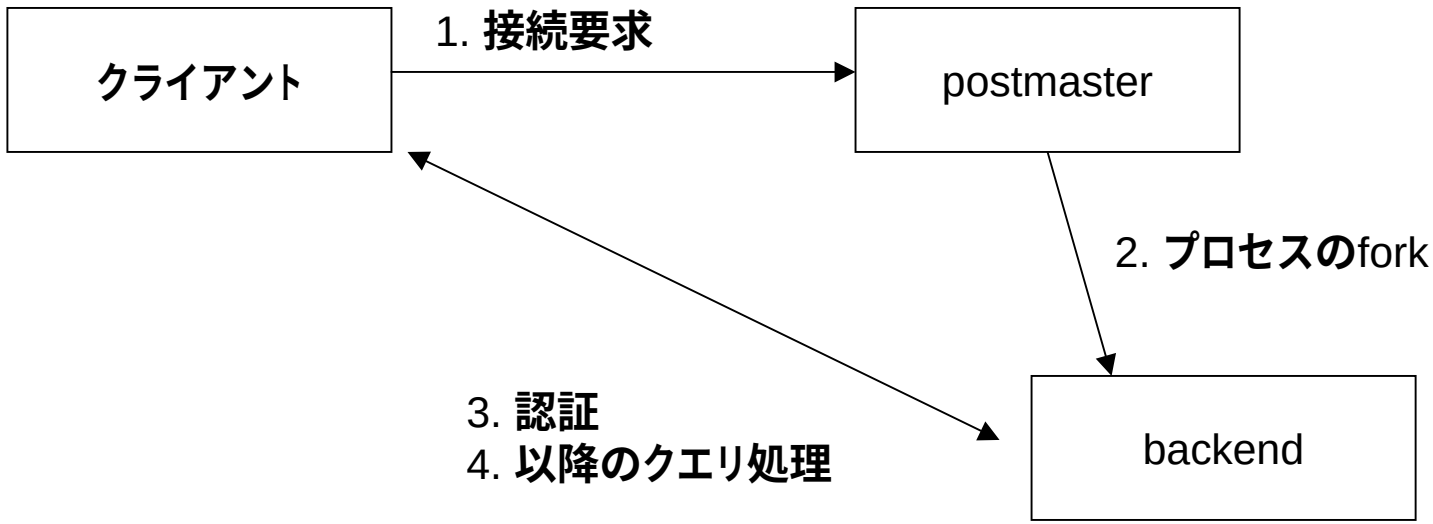
- **初期化処理**
  - **各種マネージャの初期化 (メモリマネージャ、ソケット、共有メモリ、バッファ、etc…)**
  - **ＤＢのスタートアップ処理とリカバリ**
- **フロントエンドに対する仕事**
  - **クライアントの新規接続要求の受付**
  - **クライアントからクエリのキャンセル処理の仲介**
  - pg_ctl **によるシャットダウン要求の受付**

先に説明。他は、実装のところで説明。

- **プロセス管理**
  - **バックエンドプロセスの管理**
  - check point **処理**
  - statistics collector **の監視**
- **終了処理**
  - DB **のシャットダウン処理**
- **その他**
  - **バックエンド異常終了時のクリーンナップ処理**

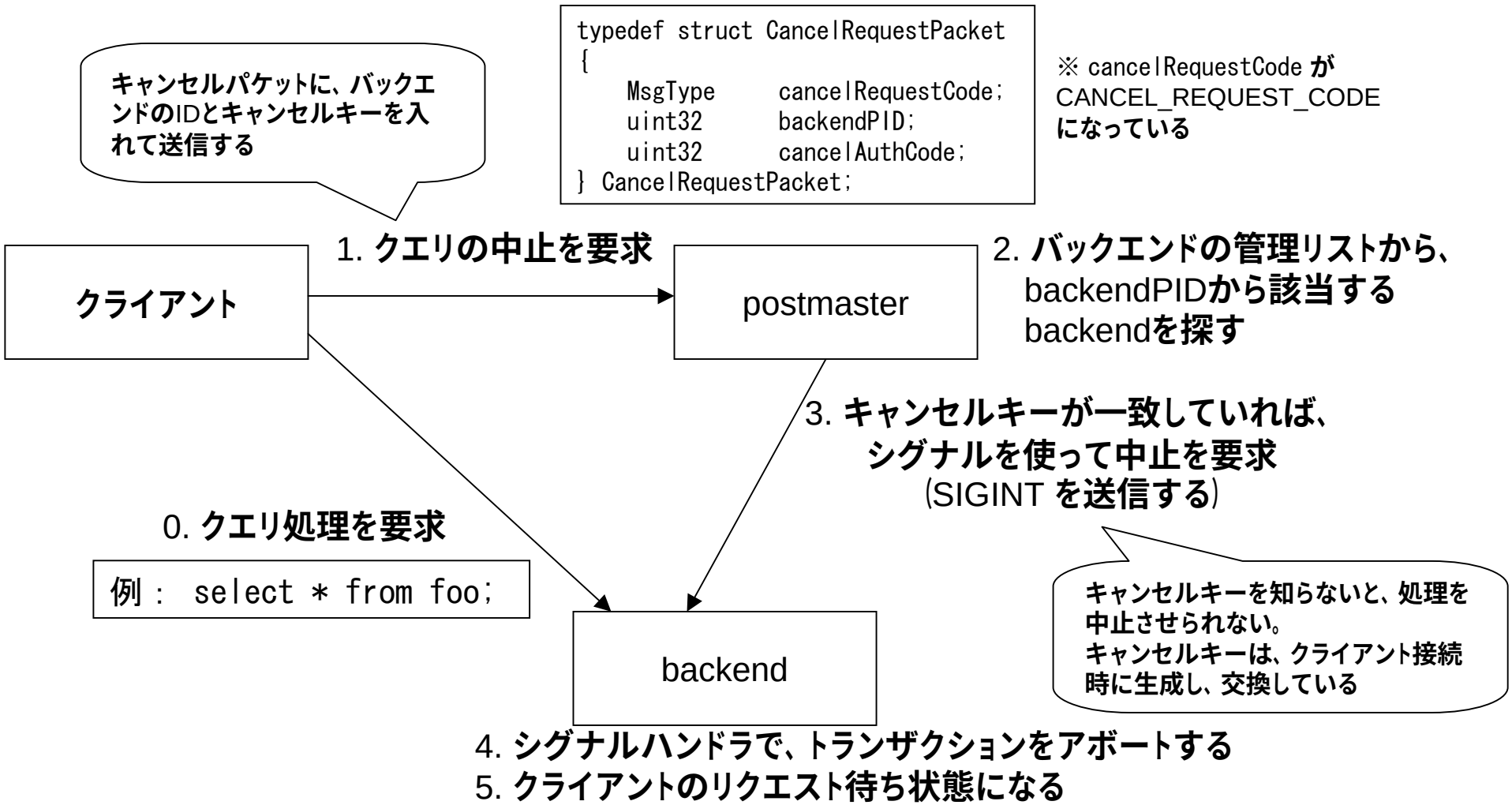
postmaster の仕事

■ クライアントの新規接続要求の受付



postmaster の仕事

■ クライアントからクエリのキャンセル処理の仲介



postmaster の実装

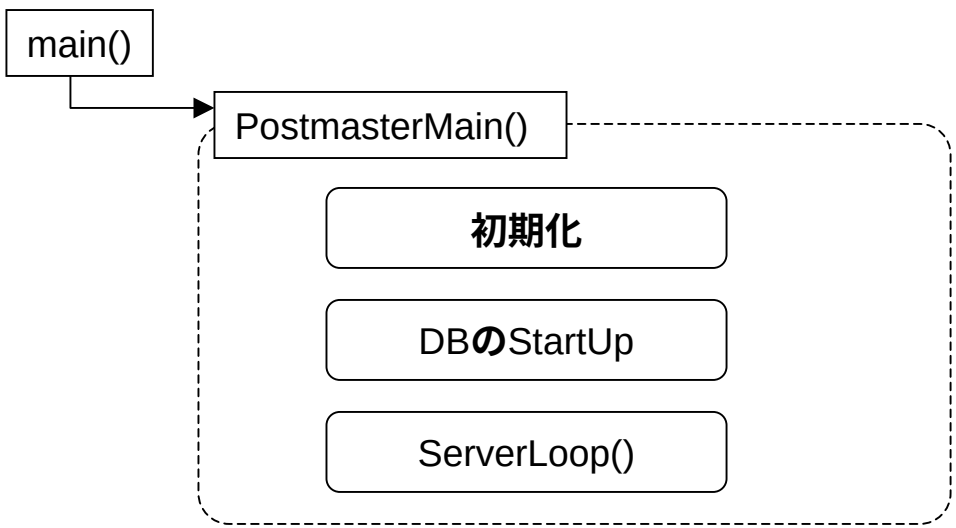
postmaster の設計思想

■ 耐障害性の設計

- ◆ フロントエンドのクライアントからのメッセージを決してブロックしないようにするため、各リソースとは、なるべく切り離されている
- ◆ なるべく子プロセスで処理を行う
  - 通常のフロントエンドのクライアントの処理は、バックエンドプロセスで行う
  - startup, shutdown, チェックポイント, 統計情報の収集などの基本的な処理でさえ、子プロセスを起動して処理する
- ◆ 共有メモリを触らない
  - 共有メモリやセマフォは、起動時に初期化は行うが、自分でそのリソースを使用することはしない
  - 若干、共有メモリを見るが、ロックを取ることはしないでいいような作りになっている
  - これによって、バックエンドの障害に引きずられない

ソースコード中で見る postmaster の処理の流れ

1. **初期化**
   - **各マネージャの初期化**
   - DB**の**startup**処理、リカバリは、子プロセスに任せる**
2. **サーバループで、各処理の振り分けを行う**
   - **クライアントの受付 (受け付けたらすぐに子プロセスを起動)**
   - **各割り込み処理の実行**
   - **チェックポイントや**shut down **処理の実行 (子プロセスを起動して行う)**



postmaster の起動シーケンス

1. –help, --version などの処理
2. メモリコンテキストの作成とコンテキストスイッチ
3. GUCとオプションの処理
4. ライブラリのプリローディング
5. デーモン化
6. DBのロック
7. ソケットの初期化
8. 共有メモリとセマフォの初期化
9. シグナルハンドラの設定
10. 統計情報収集プロセスの起動
11. クライアント認証情報の読み込み
12. DBのスタートアップ
13. サーバループ [ServerLoop()]

postmaster の起動シーケンス (ソースコードレベル)

```
main()
  ├ PostmasterMain()
  │   ├MemoryContextInit();
  │   ├InitializeGUCOptions();
  │   ├[引数の処理]
  │   ├ProcessConfigFile(PGC_POSTMASTER);
  │   ├pmdaemonize() : -S が指定された時のみ
  │   ├CreateDataDirLockFile(DataDir, true);
  │   ├reset_shared() --> CreateSharedMemoryAndSemaphores()
  │   ├[ソケットの初期化]
  │   ├[シグナルの処理]
  │   ├[統計情報収集プロセスの初期化と起動] pgstat_init();
  │   │                                    pgstat_start(); <postmaster/pgstat.c>
  │   ├StartupDataBase() --> SSDataBase(BS_XLOG_STARTUP)
  │   │                      * fork()して、子プロセスで --> BootstrapMain(ac, av);
  │   ├ServerLoop()
  │   │
  │   │
```

postmaster の起動シーケンス

1. --help, --version などの処理
   - --help または -? が第1引数だった場合、usage() を呼び出して、すぐに ExitPostmaster(0) を呼び出してプログラムを終了する
   - --version または –V が第1引数だった場合、ヘッダファイルで定義された PG_VERSION の値を出力し、すぐに ExitPostmaster(0) を呼び出してプログラムを終了する

2. メモリコンテキストの作成とコンテキストスイッチ
   - MemoryContextInit() を実行して、メモリマネージャ (utils/mmgr) の初期化を行う
   - メモリマネージャを初期化すると、最上位のメモリコンテキストTopMemoryContext が作成される
   - TopMemoryContext の下に、postmaster 用のメモリコンテキストとして、PostmasterContext を 作成する
   - PostmasterContext を作成したら、さっそく MemoryContextSwitchTo(PostmasterContext) を実行して、メモリコンテキストを切り替える

postmaster の起動シーケンス

3. GUC とオプションの処理
   - はじめにpostmaster に与えられた引数をチェックして、必要があれば GUC の初期値を上書きする
   - 次に postgres.conf ファイルを読み込んで、GUC の値を設定する

4. ライブラリのプリローディング
   - プリローディングしておくライブラリがある場合、ここで process_preload_libraries() 関数を呼び出して、ライブラリのローディングを行う
   - 特に指定されていない場合は、何もしない

postmaster の起動シーケンス

5. **デーモン化**
   - -S オプション付で起動された場合、postmaster プロセスをデーモンとして実行する
   - pmdaemonize() 関数を実行することで、これを実現している
   - postmaster をデーモンにする方法は、この関数内でプロセスを fork() して、子プロセスの方を postmaster デーモンとして処理を継続させ、親プロセスは即座に終了して処理を呼び出し元に返す
   - デーモン化されてたpostmaster プロセスは、stdin, stdout, stderr を全て /dev/null になるようにする

6. **データベースクラスタ** ($PGDATA) **のロック**
   - データディレクトリにロックファイル (postmaster.pid) を作成して、1つのデータベースクラスタに対して、複数のpostmaster プロセスが起動しないようにする

postmaster の起動シーケンス

7. **ソケットの初期化**
   - ◆ TCPIP ソケットとUNIXソケットの初期化を行う
   - ◆ TCPIPのソケットは、TCPIPの設定が有効になっている場合のみ行われる
     - デフォルトでは、TCPIPは無効になっているので、postgres.conf ファイルを変更して、TCPIP を有効にする必要がある
   - ◆ TCPIPが有効な場合でかつVirtualHost が設定してある場合、その数だけlisten するソケットを生成する
   - ◆ Unixソケットは、基本的に同一ホスト内の通信に使われる
     - ただし、JDBCは、同一ホスト内であってもTCPIPを使用する
   - ◆ Unixソケットは、/tmp に名前付きソケットが作られる

8. **共有メモリとセマフォの初期化**
   - ◆ 各モジュールの共有メモリの使用量見積もり関数にアクセスして、使用する共有メモリの量を計算し、共有メモリをまとめて確保する
   - ◆ 共有メモリを確保したら、各モジュールの共有メモリ初期化関数を呼び出す

postmaster の起動シーケンス

9. **シグナルハンドラの設定**
   - ◆ **まず、シグナルをブロックする**
   - ◆ postmaster **の使用するシグナルハンドラを設定する**
     - SIGHUP → SIGHUP_handler：**コンフィグファイルの再読込み**
     - SIGINT, SIGQUIT, SIGTERM → pmdie：postmaster **の終了**
     - SIGALRM, SIGPIPE, SIGTTIN, SIGTTOU → SIG_IGN：**無視**
     - SIGUSR1 → sigusr1_handler：**子プロセスからの通信**
     - SIGUSR2 → dummy_handler：**子プロセス起動後に使うので、ダミーとして空けておく**
     - SIGCHLD → reaper：**子プロセスの後始末など**

10. **統計情報収集プロセスの起動**
    - ◆ postmaster/pg_stats.c **で実装されている、**statistic collector **を初期化して起動する**
    - ◆ **統計情報収集のためのプロセスは、2種類起動される**

postmaster の起動シーケンス

11. **クライアント認証情報の読み込み**
    - ◆ クライアントを認証するための情報をここでファイルから読み込んでキャッシュしておく
    - ◆ 対象となるのは、次の4つのファイルである
        - hba, ident, user, group

12. DB**のスタートアップとリカバリ**
    - ◆ ここでDB の起動処理を行う
    - ◆ 先に述べたように、postmaster はこれらの処理を直接行うことはせず、子プロセスを起動して行う
    - ◆ startup のプロセスでは、コントロールファイルをチェックして、正常にシャットダウンされていない場合は、リカバリ処理を行う

13. **サーバループ** [ServerLoop()]
    - ◆ 次ページ以降で説明

DBのstartup は子プロセスで行われ、その間 postmaster のプロセスは、リカバリ処理を待たない。そのため、ServerLoop() に入るとクライアントの受付を開始しているように見える。しかし、実際は startup の子プロセスが終了して、シグナルハンドラ reaper によって StartupPID が 0 になるまで、クライアントの接続フェーズでエラーになるため接続はできない。

サーバループ

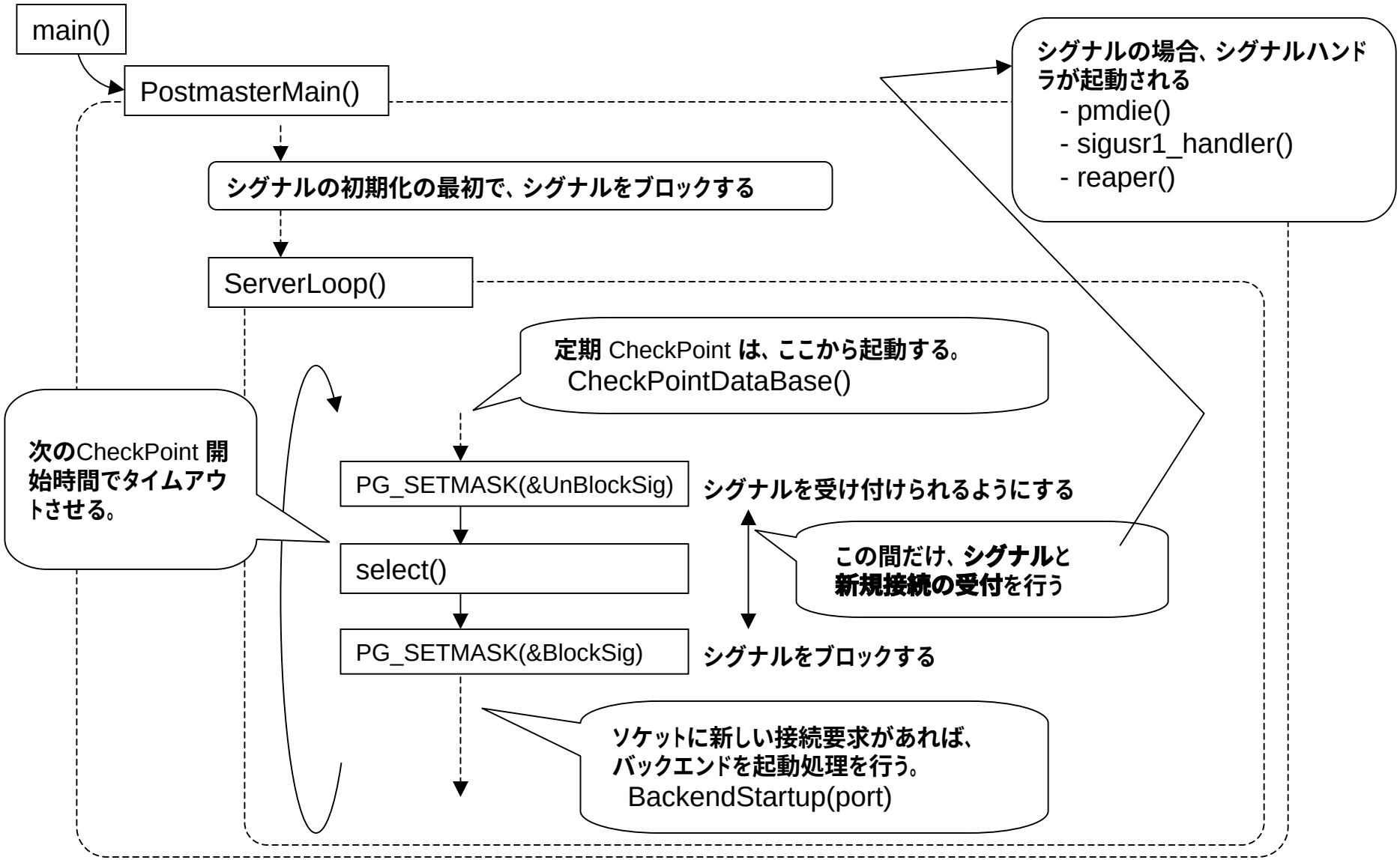
■ サーバループ [ServerLoop()] の特徴

- ◆ postmasterのメインループである
- ◆ この中でループしながら、クライアントや他のバックエンドから来る要求を処理する
- ◆ 正常終了の場合、postmaster は、この関数から return することはない
- ◆ 終了時は、proc_exit() が呼ばれて、そこから exit()する
- ◆ この関数から return されるのは、select()システムコールのエラーの場合である

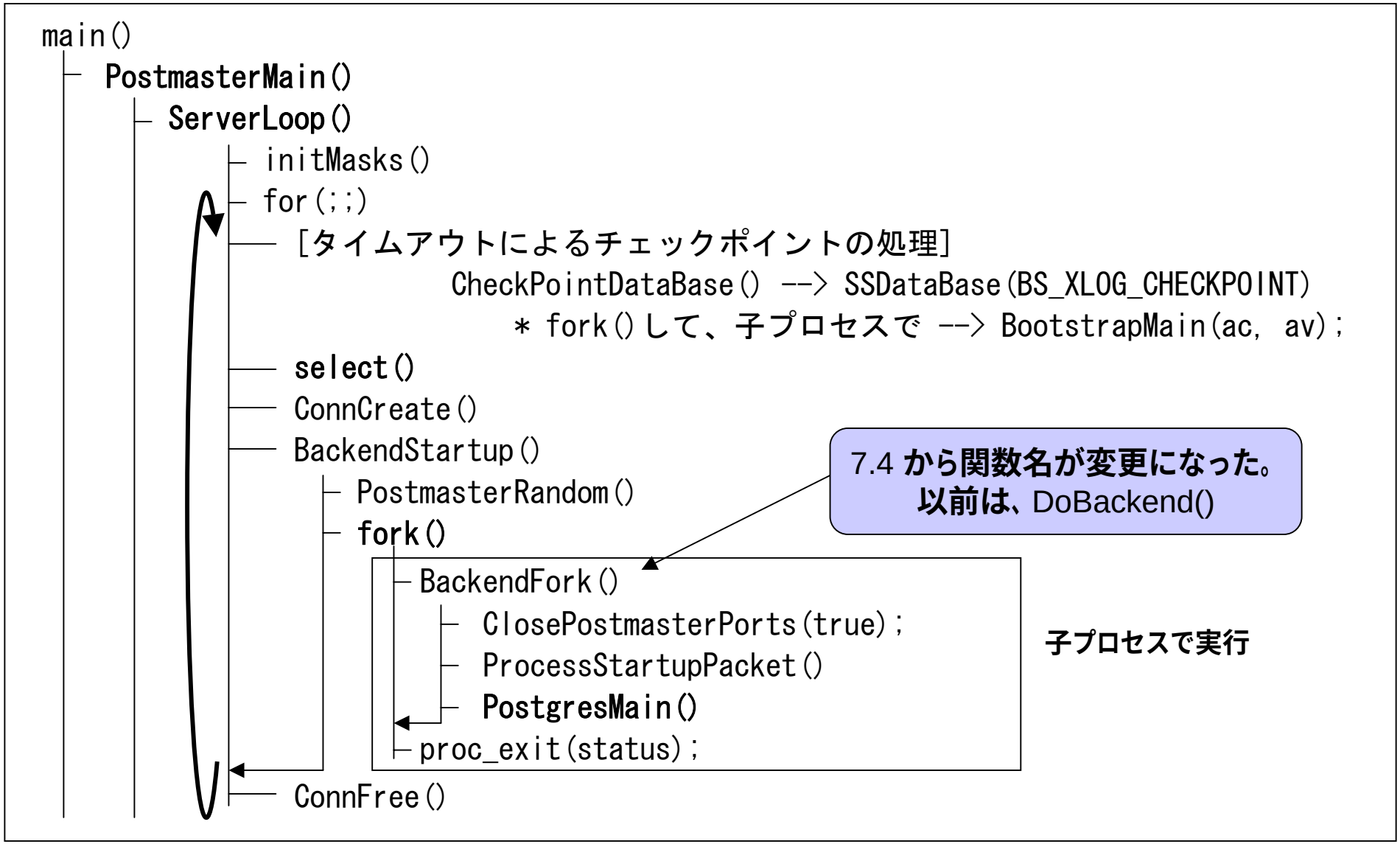
■ サーバループでの処理

- ◆ select() システムコールで、クライアントの要求を待ちつつ、check point のためのタイムアウトの時間を待つ
- ◆ select() システムコールの直前でシグナルのブロックを解除し、select() システムコール終了直後にシグナルをブロックすることで、select() システムコールで待っている間だけ、シグナルの受付を行う
- ◆ select システムコールでクライアントからの接続があった場合、プロセスを fork() して、バックエンド起動処理を行う
- ◆ 一定時間経過していたら、 子プロセスを作って check point 処理を実行させる

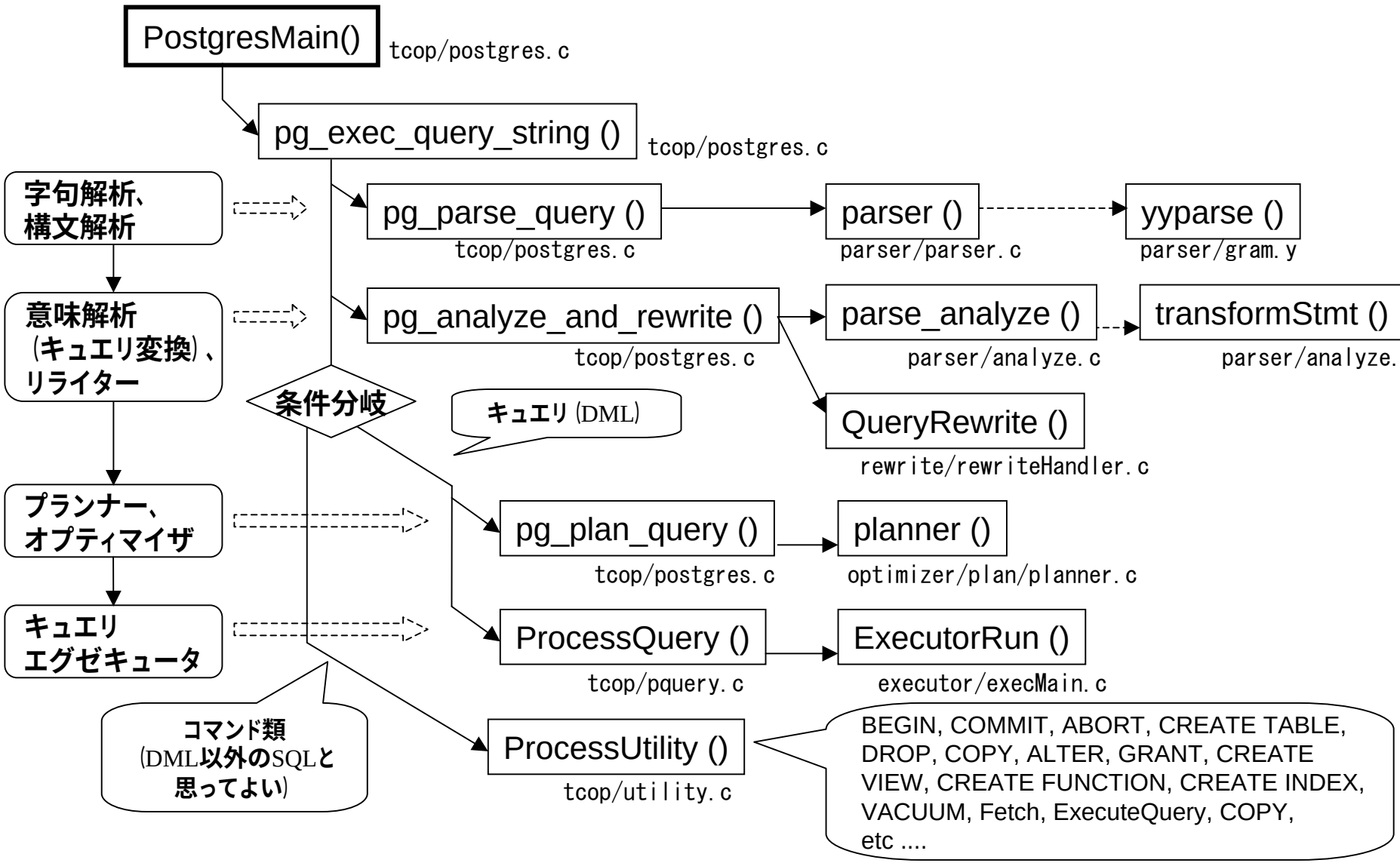
サーバループ(ソケットへの接続待ちとシグナル処理)



サーバループ（ソースコードレベル）



参考 [第1回目の資料より]：PostgresMain() からの処理



バックエンドプロセスの起動とクライアント認証

- **バックエンドとしてのfork()後、BackendFork() という関数に入る**
  - postmaster として使っていたポートを閉じる (クライアント受け付け用のソケット)
  - 自分で管理しているプロセスID、ポート番号などを新しいプロセスのものに設定
  - 終了用のシグナルハンドラを変更
    - SIGTERM, SIGQUIT → authdie：単にexit する
  - 認証に時間制限を設ける
    - このタイミングでのクライアント障害時に、リソースが解放されなくなるのを回避するため、一定時間で認証が終わらなければ、バックエンドは exit するようにする
    - SIGALRM を authdie にする
    - シグナルのブロックを再設定：postmasterの普通のブロックと違うのは、SIGALRM, SIGTERM, SIGQUIT, SIGHUP はブロックしない
    - アラームを設定する：enable_sig_alarm(AuthenticationTimeout * 1000, false)
    - アラームがあがった時点で、シグナルハンドラ authdie が起動して、exit する
  - クライアントの認証を実施：ClientAuthentication(port)
    - 認証に失敗すると、関数内でexit するため、失敗すると戻ってこない
  - アラームの設定をオフにする
  - メモリコンテキストをTopMemoryContext に移し、PostmasterContext を削除する
  - PostgresMain() を実行する

exec を呼ばずに、バックエンドのプロセスになるのが特徴

子プロセスの種類と子プロセス管理

- バックエンドプロセス
  - 通常のクライアントの処理を受け持つプロセス
  - BackendList 双方向リストで管理する
    - キャンセルキー、プロセスのID の情報を保持する

```
typedef struct bkend {
    pid_t       pid;
    long        cancel_key;
} Backend;
```

- startup, shutdown, checkpoint プロセス
  - startup処理、shutdown処理、チェックポイント処理を受け持つプロセス
  - 各処理が必要なタイミングで起動される
  - それぞれ、StartupPID, ShutdownPID, CheckPointPID という変数にプロセスIDを記録して管理する
  - チェックポイントだけは、 BackendList リストにも入っている
- 統計情報収集プロセス
  - 次の2つのプロセスである
    - postgres: stats buffer process
    - postgres: stats collector process
  - postmaster としては、起動だけ行ってあとは何もしない
    - ServerLoop() の最後で、何らかの理由で統計情報収集プロセスが実行中でない場合、スタートさせる処理は入っている

チェックポイント実行

■ チェックポイントとは？ (今回の本題でないので、簡単な説明)

- ◆ DBは、障害時に、トランザクションログを適用することで、リカバリが可能である
- ◆ 物理ロギングをすることにより、Idempodent (べき等性)を保証する
  - べき等とは、同じ操作を何度繰り返しても、同じ結果になること。つまり、同じログを何度上書きしても同じ結果になる。また、元のデータが更新前の値か後の値かに関わらず、更新後の処理を適用していいことになる。
  - PostgreSQL の場合、物理ロギングと言っても、ページに対する操作は論理的なものであり、ページ内のデータに対して物理的なものであるので、物理論理ロギングと呼ばれる(これでもべき等性は保証される)
- ◆ リカバリ時には、どこからログを適用するべきかわからないので、最悪、ログの先頭からログを適用する必要がある (リカバリに時間がかかる)
- ◆ チェックポイントとは、リカバリの開始点を決定するための仕組み
- ◆ チェックポイントにより、あるポイントより古いログを捨ててもリカバリが可能になる
  - チェックポイント処理が完了した時点で、チェックポイント開始点より前のログは不要になる
- ◆ チェックポイントは、その処理中にバッファ中のダーティページをディスクに書き出す
  - バッファの書き出しは、Write Ahead Logging のルールに従う

チェックポイント実行の2つのタイミング

■ タイムアウトによるチェックポイントの実行

- ◆ サーバループで、select() システムコールに、checkpoint_timeoutまでの残り時間をタイムアウトとして設定し、select() システムコールを実行する
- ◆ select() システムコールは、クライアントからの接続でも抜けるため、checkpoint_timeout の時間が経過したかチェックする
  - checkpoint_timeout の時間が経過していたら子プロセスを起動してチェックポイントを実行する
  - checkpoint_timeout の時間が経過していたいなかったら、前回チェックポイントを実行した時間から現在までの経過時間の差を、checkpoint_timeout から引いた時間を計算し、それを新しく select() システムコールのタイムアウト時間とする

■ バックエンドからの通知によるチェックポイントの実行

- ◆ xlogセグメントが一定量に達したら、バックエンドが共有メモリのフラグエリアにフラグをセットし、SIGUSR1 を送ってくる
- ◆ postmaster は、シグナルハンドラ sigusr1_hander() の中で、チェックポイント実行のフラグを検出して、チェックポイントを実行する

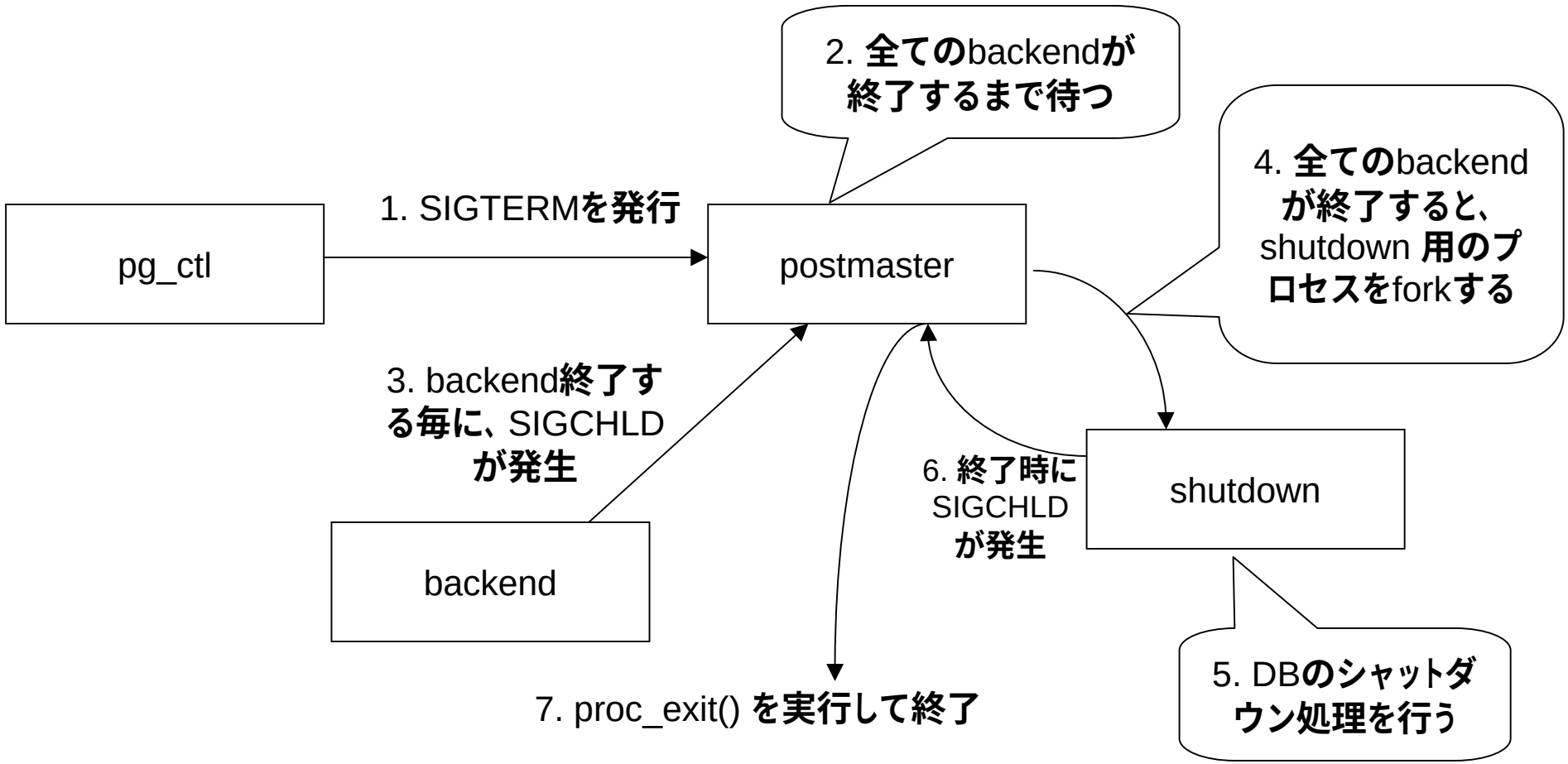
チェックポイント実行後の処理

■ チェックポイントの後始末

- ◆ チェックポイントは子プロセスで実行される
- ◆ チェックポイントが終了すると子プロセスは終了するので、SIGCHLD が postmaster に通知される
- ◆ シグナルハンドラ reaper() で、終了した子プロセスのプロセスID をチェックして、チェックポイント用の子プロセスが終了したことを検出する
- ◆ postmasrterのグローバル変数 CheckPointPID を 0 にする

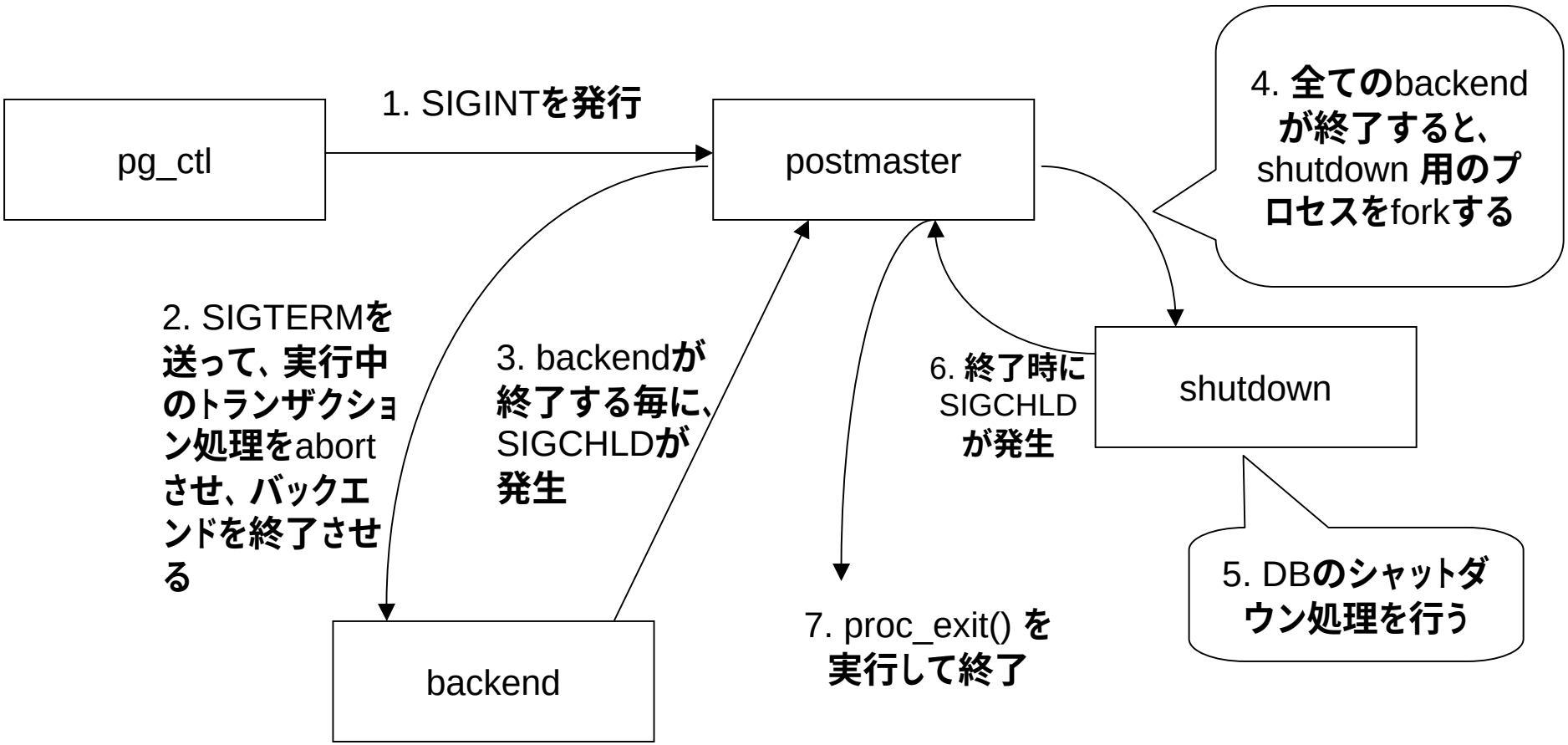
DBのシャットダウン

■ pg_ctl [-m smart] stop **によるシャットダウン要求**



DBのシャットダウン

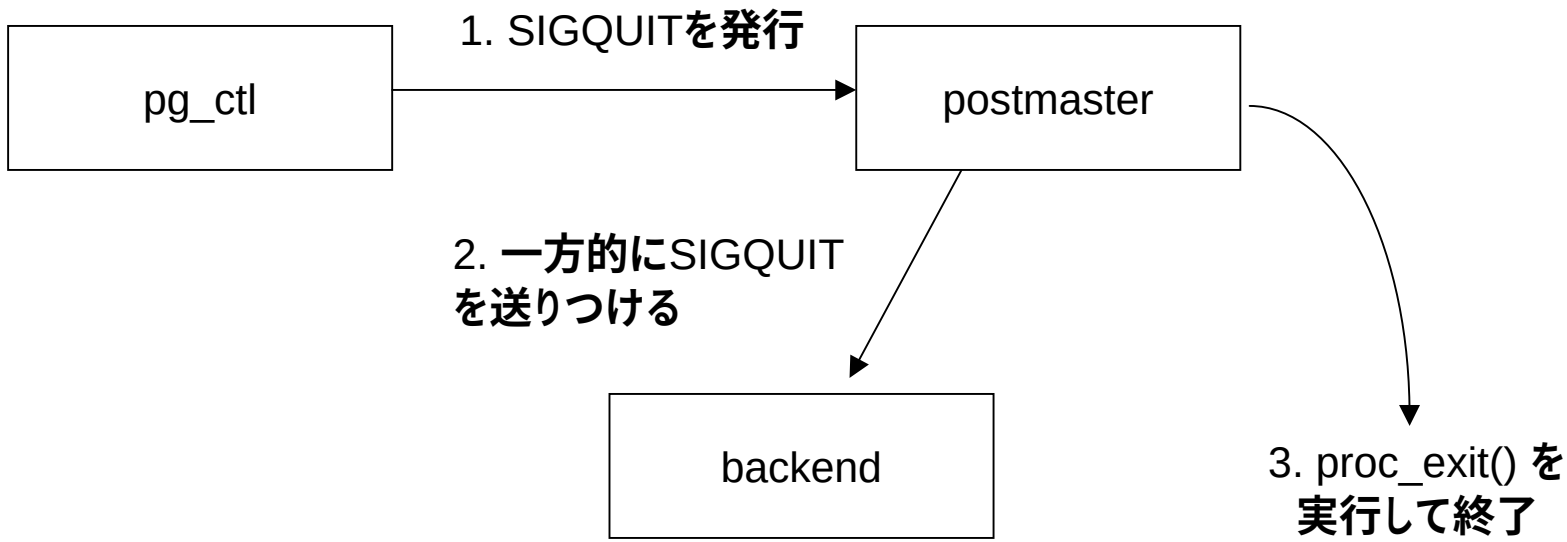
■ pg_ctl –m fast stop によるシャットダウン要求



※ backend は、きりがいいところで、自発的にアボート処理を行う。
従って、いつまで経ってもpostmaster が終了できないことがある。

DBのシャットダウン

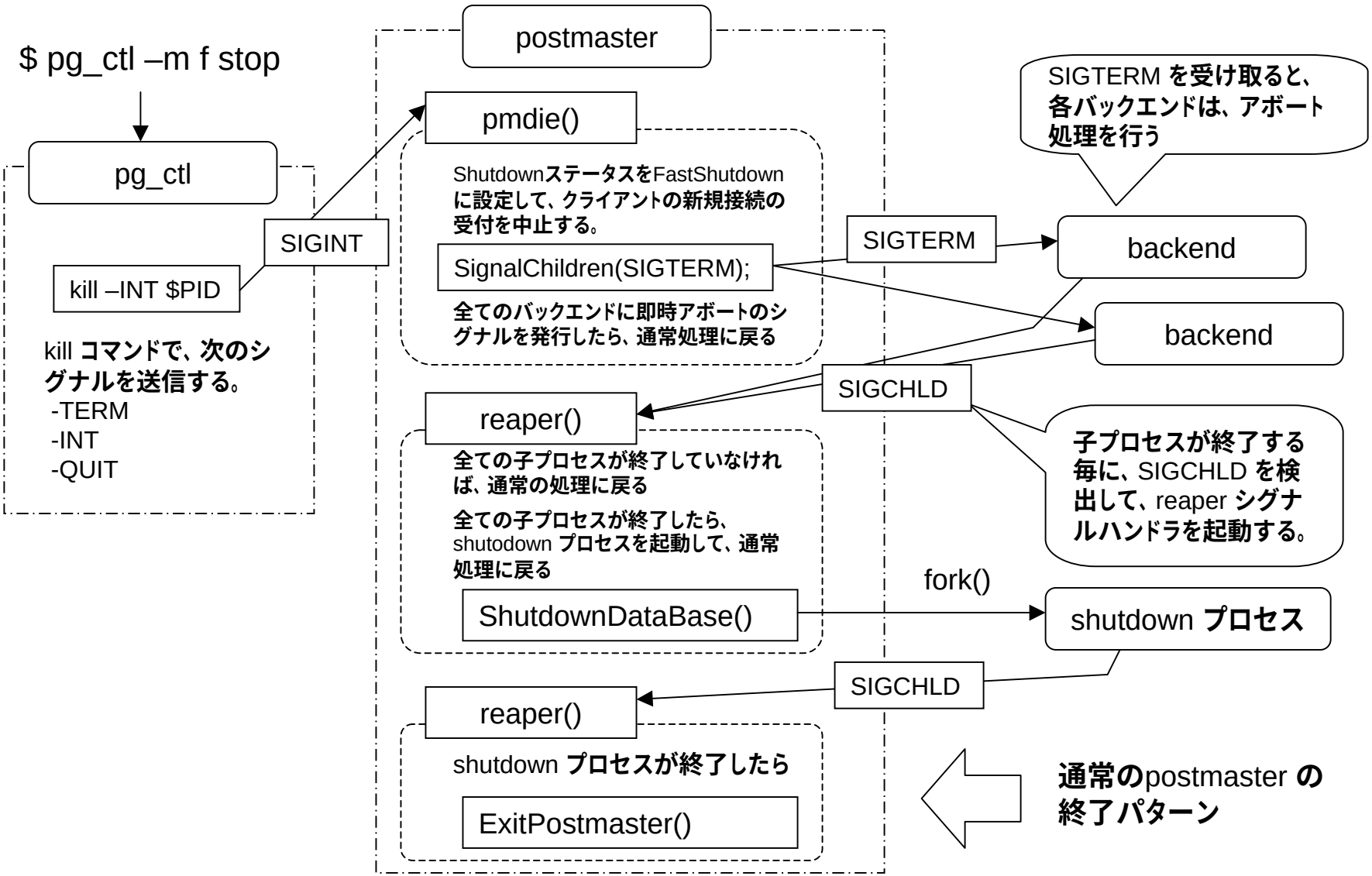
■ pg_ctl –m immediate stop によるシャットダウン要求



SIGQUIT による終了では、ＤＢのシャットダウン処理は行わない。
proc_exit() を実行することで共有メモリまわりの後始末を最低限行う

※ SIGKILL を直接送ってpostmaster を停止させた場合、共有メモリ関連の後始末がされないので、なるべくOSを再起動したほうがよい。（一応、再初期化をして続行はできるようになっている？）

DBのシャットダウン



バックエンド異常終了時のクリーンナップ処理

■ バックエンドの異常終了を検出すると…

- ◆ FatalError という変数がtrueになる
- ◆ これ以上、新規クライアントの接続はできなくなる
- ◆ シグナルハンドラ reaper() で処理する
- ◆ 全てのクライアントの処理が終了してから処理を開始
  - 全てのクライアントが終了しないとreaperは、クリーンナップ処理をしないで、シグナルハンドラは終了する
- ◆ クリーンナップの処理内容は、
  - 共有メモリを初期化
  - StartupDataBase を実行

■ バックエンドの異常終了の検出のタイミング

- ◆ SIGCHLDが送られてきて、シグナルハンドラ reaper() が起動される
- ◆ reaper() の中で、通常のバックエンドプロセスだと判断したときに行う終了処理 CleanupProc(pid, exitstatus) に入る
- ◆ exitstatus を調べて、異常終了の場合、グローバル変数FatalError をtrueにする

BootstrapMain() への移行

- 子プロセスのうち、startup, shutdown, チェックポイントの処理は、子プロセスを呼び出すが、postmaster からの切り替えの処理が終了した時点で、bootstrap.c に定義してある BootstrapMain() を呼び出すことになる
  - BootstrapMain() を呼ぶ前に行うのは、stdout, stderr のフラッシュと、プロセス名の設定
    - “startup subprocess“, "shutdown subprocess“, "checkpoint subprocess“
  - BootstrapMain() の引数として、startup は –x2、 shutdown は –x4、チェックポイントは –x3 を渡す

- BootstrapMain() では、いろいろな初期化を行う関数があるが、postmaster から実行された場合は、ほとんど何もしない
  - これらは、initdb から postmaster –boot で呼び出された時用

BootstrapMain() (1/2)

startup、shutdown、チェックポイントで基本となるのは、それぞれ次の関数

- startup
  - StartupXLOG()
    - コントロールファイルをチェックして、正常にシャットダウンされていない場合、リカバリ処理を行う
    - StartUpID をインクリメントする
  - LoadFreeSpaceMap()：各テーブルの空き領域情報の読み出し (7.4より)

- shutdown
  - ShutdownXLOG()
    - チェックポイントを実行する：CreateCheckPoint() は、shutdown 時は、コントロールファイにDB_SHUTDOWNED のステータスを書き込む
    - CLOG のフラッシュを行う
  - DumpFreeSpaceMap()：各テーブルの空き領域情報の書き出し (7.4より)

BootstrapMain()（2/2）

■ チェックポイント

- CreateDummyCaches()
- CreateCheckPoint(false, false)：**チェックポイントの実行**
- SetSavedRedoRecPtr()：postmaster **に** REDO **の位置を渡す**
  - **コントロールファイルには、**CreateCheckPoint()**の中で書いている**

ここまでの各処理は、これらを実行すると、proc_exit(0) **を呼び出してプロセスを終了する。**

■ おまけ (BOOTSTRAP**時**)

- initdb **から** postgres –boot –x1 **として呼ばれる**
- BootStrapXLOG()
  - pg_control**ファイルの作成、**xlog**ファイル の作成、**clog **の作成などが行われる**
- StartupXLOG()：**リカバリはないので、初期化として呼ばれている**

おしまい