Signal Handling

Andres Freund

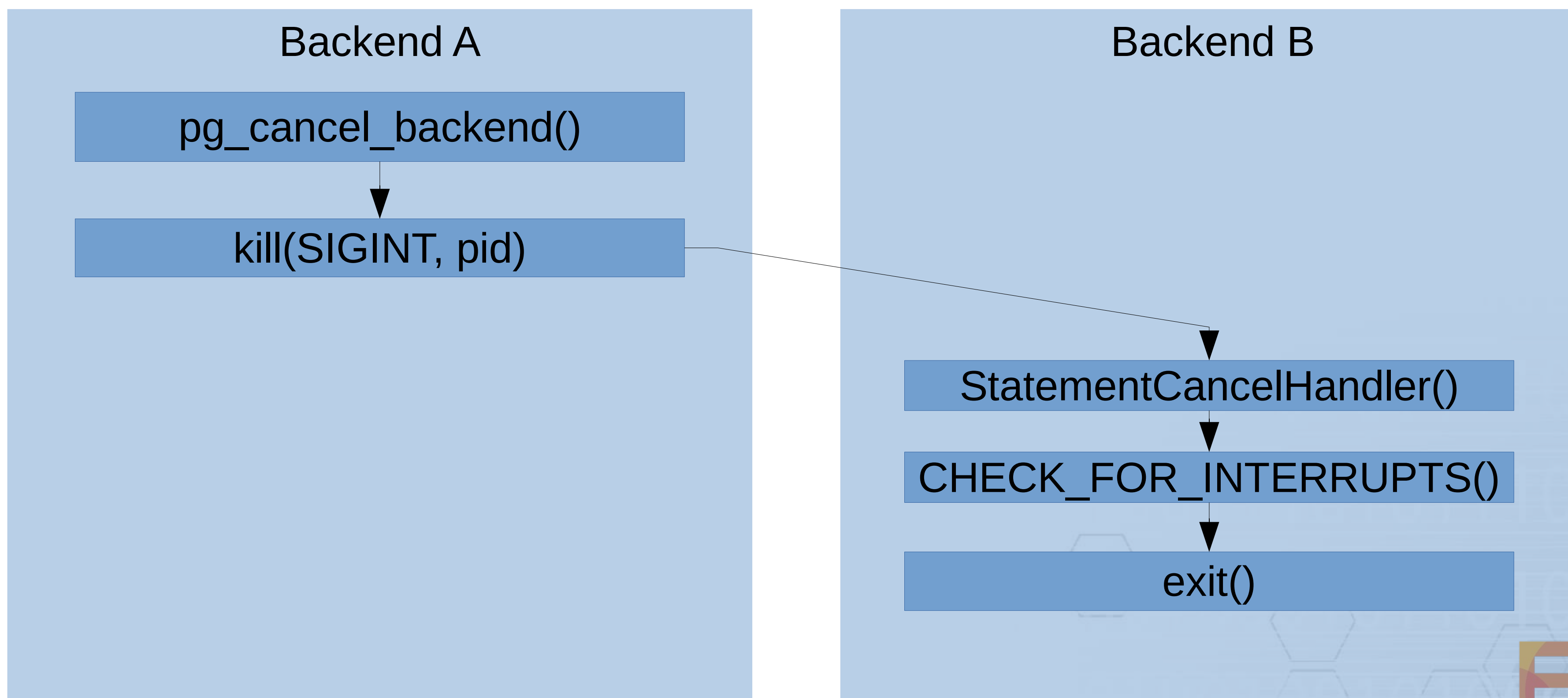
PostgreSQL Developer & Committer

Email: andres@anarazel.de
Email: andres.freund@enterprisedb.com
Twitter: @AndresFreundTec

anarazel.de/talks/2019-02-12-bangalore-edb-office/signals.pdf

What are signals?

- inter process communication
- SIGKILL, SIGINT, SIGHUP, …



EDB POSTGRES

How do signals work?

- Default signal handlers / behaviour
  - exit, ignore, dump core
  - not all can be overridden
- Signals interrupt running code
- Registered handler for signal gets called
- registered with sigaction() (in postgres pqsignal)
- Asynchronous, collapsible
- http://man7.org/linux/man-pages/man7/signal.7.html
- http://man7.org/linux/man-pages/man2/sigaction.2.html

EDB POSTGRES

First the signals described in the original POSIX.1-1990 standard.

Signal	Value	Action	Comment
SIGHUP	1	Term	Hangup detected on controlling terminal or death of controlling process
SIGINT	2	Term	Interrupt from keyboard
SIGQUIT	3	Core	Quit from keyboard
SIGILL	4	Core	Illegal Instruction
SIGABRT	6	Core	Abort signal from abort(3)
SIGFPE	8	Core	Floating-point exception
SIGKILL	9	Term	Kill signal
SIGSEGV	11	Core	Invalid memory reference
SIGPIPE	13	Term	Broken pipe: write to pipe with no readers; see pipe(7)
SIGALRM	14	Term	Timer signal from alarm(2)
SIGTERM	15	Term	Termination signal
SIGUSR1	30,10,16	Term	User-defined signal 1
SIGUSR2	31,12,17	Term	User-defined signal 2
SIGCHLD	20,17,18	Ign	Child stopped or terminated
SIGCONT	19,18,25	Cont	Continue if stopped
SIGSTOP	17,19,23	Stop	Stop process
SIGTSTP	18,20,24	Stop	Stop typed at terminal
SIGTTIN	21,21,26	Stop	Terminal input for background process
SIGTTOU	22,22,27	Stop	Terminal output for background process

The signals **SIGKILL** and **SIGSTOP** cannot be caught, blocked, or ignored.

```
static void
reload_config_handler(SIGNAL_ARGS)
{
    elog(DEBUG1, "reloading config now");
    LWLockAcquire(ConfigLock, LW_EXCLUSIVE);
    ProcessConfigFile(PGC_SIGHUP);
    LWLockRelease(ConfigLock);
}
```

Errno

```c
static void
reload_config_handler(SIGNAL_ARGS)
{
    elog(DEBUG1, "reloading config now");
    LWLockAcquire(ConfigLock, LW_EXCLUSIVE);
    ProcessConfigFile(PGC_SIGHUP);
    LWLockRelease(ConfigLock);
}
```

```c
    errno = 0;
    pgstat_report_wait_start(WAIT_EVENT_REORDER_BUFFER_WRITE);
    if (write(fd, rb->outbuf, ondisk->size) != ondisk->size)
    {
        int         save_errno = errno;

        CloseTransientFile(fd);

        /* if write didn't set errno, assume problem is no disk space */
        errno = save_errno ? save_errno : ENOSPC;
        ereport(ERROR,
                (errcode_for_file_access(),
                 errmsg("could not write to data file for XID %u: %m",
                        txn->xid)));
    }
```

EDB POSTGRES

How to write safe signal handlers

- Only use async signal safe system functions
  - http://man7.org/linux/man-pages/man7/signal-safety.7.html
- Ensure handler is reentrant
  - Access no (as little as possible) shared state
    - locks
    - any data structure more complicated than a boolean
  - Only call reentrant functions
    - i.e. functions that only depend on data passed in as parameters
- Never block
- Use latches
- Mask signals that can’t be handled (in normal control flow or handler)
  - PG_SETMASK(&BlockSig) / PG_SETMASK(&UnBlockSig)
  - signals will be run once unblocked

EDB™ POSTGRES

```c
extern PGDLLIMPORT volatile sig_atomic_t ConfigReloadPending;

/*
 * SIGHUP: set flag to re-read config file at next convenient time.
 *
 * Sets the ConfigReloadPending flag, which should be checked at convenient
 * places inside main loops. (Better than doing the reading in the signal
 * handler, ey?)
 */
void
PostgresSigHupHandler(SIGNAL_ARGS)
{
    int         save_errno = errno;

    ConfigReloadPending = true;
    SetLatch(MyLatch);

    errno = save_errno;
}
```

EDB POSTGRES

```c
    /* If we have pending write here, go to slow path */
    for (;;)
    {
        int         wakeEvents;
        long        sleeptime;

        /* Check for input from the client */
        ProcessRepliesIfAny();

        /* die if timeout was reached */
        WalSndCheckTimeOut();

        /* Send keepalive if the time has come */
        WalSndKeepaliveIfNecessary();

        if (!pq_is_send_pending())
            break;

        sleeptime = WalSndComputeSleeptime(GetCurrentTimestamp());

        wakeEvents = WL_LATCH_SET | WL_EXIT_ON_PM_DEATH |
            WL_SOCKET_WRITEABLE | WL_SOCKET_READABLE | WL_TIMEOUT;

        /* Sleep until something happens or we time out */
        (void) WaitLatchOrSocket(MyLatch, wakeEvents,
                                 MyProcPort->sock, sleeptime,
                                 WAIT_EVENT_WAL_SENDER_WRITE_DATA);

        /* Clear any already-pending wakeups */
        ResetLatch(MyLatch);

        CHECK_FOR_INTERRUPTS();

        /* Process any requests or signals received recently */
        if (ConfigReloadPending)
        {
            ConfigReloadPending = false;
            ProcessConfigFile(PGC_SIGHUP);
            SyncRepInitConfig();
        }

        /* Try to flush pending output to the client */
        if (pq_flush_if_writable() != 0)
            WalSndShutdown();
    }

    /* reactivate latch so WalSndLoop knows to continue */
    SetLatch(MyLatch);
}
```

EDB POSTGRES™

Latches

- NOT the latches from database literature
- src/include/storage/latch.h – has a good introductory comment
- Allows to wait for:
  - latch being set by signal handler / other process
  - timer
  - network IO (multiple connections if necessary)
  - postmaster death

EDB™ POSTGRES

Common Signals

- SIGKILL – kill without chance for cleanup
- SIGSTOP / CONT – suspend, continue execution
- SIGINT – interrupt program (often quit or cancel current action)
  - in postgres: cancel query
- SIGTERM – quit
  - in postgres: terminate connection
- SIGQUIT – really quit (and dump core)
  - in postgres: panic shutdown
- SIGABRT – dump core and quit
- SIGALRM – timer is up
  - in postgres: timeout handling via src/include/utils/timeout.h
- SIGUSR1/2 – user defined
  - in postgres: 1: latches, procsignals 2: autovacuum, startup process, ...

EDB POSTGRES™

```c
/*
 * Reasons for signalling a Postgres child process (a backend or an auxiliary
 * process, like checkpointer).  We can cope with concurrent signals for different
 * reasons.  However, if the same reason is signaled multiple times in quick
 * succession, the process is likely to observe only one notification of it.
 * This is okay for the present uses.
 *
 * Also, because of race conditions, it's important that all the signals be
 * defined so that no harm is done if a process mistakenly receives one.
 */
typedef enum
{
    PROCSIG_CATCHUP_INTERRUPT,  /* sinval catchup interrupt */
    PROCSIG_NOTIFY_INTERRUPT,   /* listen/notify interrupt */
    PROCSIG_PARALLEL_MESSAGE,   /* message from cooperating parallel backend */
    PROCSIG_WALSND_INIT_STOPPING,   /* ask walsenders to prepare for shutdown  */

    /* Recovery conflict reasons */
    PROCSIG_RECOVERY_CONFLICT_DATABASE,
    PROCSIG_RECOVERY_CONFLICT_TABLESPACE,
    PROCSIG_RECOVERY_CONFLICT_LOCK,
    PROCSIG_RECOVERY_CONFLICT_SNAPSHOT,
    PROCSIG_RECOVERY_CONFLICT_BUFFERPIN,
    PROCSIG_RECOVERY_CONFLICT_STARTUP_DEADLOCK,

    NUM_PROCSIGNALS             /* Must be last! */
} ProcSignalReason;

/*
 * prototypes for functions in procsignal.c
 */

extern int SendProcSignal(pid_t pid, ProcSignalReason reason,
               BackendId backendId);
```