

Signals, Latches, and Interrupts

Past and future of inter-process communication in
PostgreSQL

This presentation has two parts:

1. Current mechanism
2. A new “interrupts” mechanism I’ve been working on
 - Not committed yet
 - Hoping for version 20
 - [pgsql-hackers thread](#): "Interrupts vs signals"

Inter-process communication

- Locking
- Shared memory, atomics
- Launching parallel worker processes
- Message queues
- Waiting for another process to do something
- Waking up another process

Inter-process communication

- Locking
- Shared memory, atomics
- Launching parallel worker processes
- Message queues
- **Waiting for another process to do something**
- **Waking up another process**

Operating system primitives available

- POSIX / SysV Shared memory
- POSIX / SysV Semaphores
- Unix signals
- Pipes, sockets
- Linux futexes
- pthread condition variables, mutexes
- Windows

Semaphores, locks

- Each backend process has one semaphore to sleep on
- Used to wake up a process waiting on an LWLock
 - and a few other little things
- Impossible to wait for a semaphore and a signal or other wakeup at the same time

Unix Signals are used for most wakeups

- But there are abstractions on top of them
- There is only a limited number of signals
 - Multiplexing

How Unix Signals are used in PostgreSQL

Signals accepted by postmaster:

- SIGTERM: Shut down (smart)
- SIGINT: Shut down (fast)
- SIGQUIT: Shut down (immediate)
- SIGHUP: Reload config file

You can send these directly from the shell:

```
kill -SIGHUP <postmaster pid>
```

or

```
pg_ctl kill HUP <postmaster pid>
```

How Unix Signals are used in PostgreSQL

Signals accepted by backends:

- **SIGTERM: Terminate backend gracefully**
 - `pg_terminate_backend()`
 - Sent by postmaster on smart or fast shutdown
- **SIGQUIT: Immediate shutdown**
 - Sent by postmaster on immediate shutdown
 - causes crash recovery on restart
- **SIGHUP: Reload config file**
- **SIGINT: Cancel query**
 - `pg_cancel_backend()`, CTRL-C in psql

Signals used internally between processes

- SIGUSR1:
 - backends: multiplexed for ProcSignals
 - postmaster: multiplexed for PMSignals
 - Syslogger: rotate log
- SIGUSR2:
 - Wakeups between autovacuum launcher and workers
 - Wake up WAL archiver
 - Checkpointer: shut down
 - I/O workers: shut down
 - Startup process: promote standby

Signals used internally within one process

- SIGALRM: Used by timers
 - `statement_timeout`, etc.

ProcSignal multiplexing

- SIGUSR1 is multiplexed for many purposes
- Each backend has an array of "reason flags" in shared memory
- To send a procsignal:
 1. Sets the target backend's reason bit
 2. Send SIGUSR1 to wake it up

ProcSignal

Sender:

1. Set a flag in the receiver's array in shared memory
2. Send SIGUSR1

Receiver signal handler:

3. Check the reason bit in shared memory
4. Set corresponding global variable
5. SetLatch(MyLatch);

Receiver CHECK_FOR_INTERRUPTS():

6. Check the global variable
7. Call the right procsignal "handler" function

ProcSignal multiplexing

[illegible]

Postmaster is special

On a backend crash, the postmaster process must be able to kill other processes and restart the system.

- It does not participate in any locking with other processes
- It does not trust the contents of shared memory

-> Postmaster cannot use ProcSignals, only plain Unix signals

PMSignal multiplexing

```
typedef enum
{
    PMSIGNAL_RECOVERY_STARTED,          /* recovery has started */
    PMSIGNAL_RECOVERY_CONSISTENT,        /* recovery has reached consistent state */
    PMSIGNAL_BEGIN_HOT_STANDBY,          /* begin Hot Standby */
    PMSIGNAL_ROTATE_LOGFILE,             /* send SIGUSR1 to sysloger to rotate logfile */
    PMSIGNAL_START_AUTOVAC_LAUNCHER,     /* start an autovacuum launcher */
    PMSIGNAL_START_AUTOVAC_WORKER,       /* start an autovacuum worker */
    PMSIGNAL_IO_WORKER_GROW,             /* I/O worker pool wants to grow */
    PMSIGNAL_BACKGROUND_WORKER_CHANGE,   /* background worker state change */
    PMSIGNAL_START_WALRECEIVER,          /* start a walreceiver */
    PMSIGNAL_ADVANCE_STATE_MACHINE,      /* advance postmaster's state machine */
    PMSIGNAL_XLOG_IS_SHUTDOWN,           /* ShutdownXLOG() completed */
} PMSignalReason;
```

Doing stuff in signal handler – bad idea!

- Long time ago, signals were processed directly in the signal handler
 - Still some remnants of that: SIGTERM during authentication
- Nowadays, every signal handler just sets a global variable, which is checked later
- CHECK_FOR_INTERRUPTS() calls are sprinkled all over the codebase
 - Especially in any loops

Latches, WaitEventSets

- WaitEventSet is an OS-agnostic abstraction to wait for:
 - Socket to become readable / writable
 - Latch to be set by signal handler or another process
- Fixes the classic race condition waiting on a signal
 - pselect() / ppoll()
 - "self-pipe trick"
 - OS specific mechanisms:
 - signalfd on Linux
 - kqueue on FreeBSD
 - Windows

Typical wait-loop using a Latch

```
for (;;)
{
    ResetLatch(MyLatch);

    CHECK_FOR_INTERRUPTS();

    if (...)
        break;

    (void) WaitLatch(MyLatch,
                    WL_LATCH_SET | WL_EXIT_ON_PM_DEATH,
                    0, WAIT_EVENT_...);
}
```

Latch drawbacks

- Only one latch for each process
 - Some exceptions: startup process latch for promotions
- Can only wait for one Latch at a time

recoveryWakeupLatch

commit ac22929a2613e122708bd0172508ac863c51c1cc

Author: Fujii Masao <fujii@postgresql.org>

Date: Wed Nov 4 16:41:29 2020 +0900

Get rid of the dedicated latch for signaling the startup process.

This commit gets rid of the dedicated latch for signaling the startup process in favor of using its procLatch, since that comports better with possible generic signal handlers using that latch.

Commit 1e53fe0e70 changed background processes so that they use standard SIGHUP handler. Like that, this commit also makes the startup process use standard SIGHUP handler to simplify the code.

Author: Fujii Masao

Reviewed-by: Bharath Rupireddy, Michael Paquier

Discussion: https://postgr.es/m/CALj2ACXPorUqePswDtOeM_s82v9RW32E1fYmOPZ5NuE+TWKj_A@mail.gmail.com

nope

commit 00f690a239932e477f25120d19b08aacdc30deb7

Author: Fujii Masao <fujii@postgresql.org>

Date: Thu Dec 17 18:06:51 2020 +0900

Revert "Get rid of the dedicated latch for signaling the startup process".

Revert ac22929a26, as well as the followup fix 113d3591b8. Because it **broke the assumption that the startup process waiting for the recovery conflict on buffer pin should be waken up only by buffer unpin or the timeout** enabled in ResolveRecoveryConflictWithBufferPin(). **It caused, for example, SIGHUP signal handler or walreceiver process to wake that startup process up unnecessarily frequently.**

Additionally, add the comments about why that dedicated latch that the reverted patch tried to get rid of should not be removed.

Thanks to Kyotaro Horiguchi for the discussion.

Author: Fujii Masao

Discussion: <https://postgr.es/m/d8c0c608-021b-3c73-fffd-3240829ee986@oss.nttdata.com>

recoveryWakeLatch

- Whenever a standby receives WAL from the primary, the startup process is woken up by setting it's latch
- If recovery conflicts with a query in the standby, the startup process pauses to wait for the query to finish
- While it's paused, the WAL receiver still continuously wakes up the startup process, even though it's waiting for the query, not for more WAL to arrive

CHECK_FOR_INTERRUPTS() drawbacks

- Hard-coded set of "flags" it checks
- Not extensible
- Difficult to process only some interrupts or mask only some interrupts

In ProcessInterrupts(), i.e CHECK_FOR_INTERRUPTS()

```
/*
 * Don't allow query cancel interrupts while reading input from the
 * client, because we might lose sync in the FE/BE protocol. (Die
 * interrupts are OK, because we won't read any further messages from the
 * client in that case.)
 *
 * See similar logic in ProcessRecoveryConflictInterrupts().
 */
if (QueryCancelPending && QueryCancelHoldoffCount != 0)
{
    /*
     * Re-arm InterruptPending so that we process the cancel request as
     * soon as we're done reading the message. (XXX this is seriously
     * ugly: it complicates INTERRUPTS_CAN_BE_PROCESSED(), and it means we
     * can't use that macro directly as the initial test in this function,
     * meaning that this code also creates opportunities for other bugs to
     * appear.)
     */
    InterruptPending = true;
}
```

NEW: Interrupts

From: Thomas Munro <thomas(dot)munro(at)gmail(dot)com>
To: pgsql-hackers <pgsql-hackers(at)postgresql(dot)org>
Subject: Interrupts vs signals
Date: 2021-10-20 18:55:54

Hi,

I wonder if we really need signals to implement interrupts. Given that they are non-preemptive/cooperative (work happens at the next CFI()), why not just use shared memory flags and latches? That skips a bunch of code, global variables and scary warnings about programming in signal handlers.

I sketched out some code to try that a few months back, while speculating about bite-sized subproblems that would come up if each backend is, one day, a thread.

There are several other conditions that are also handled by `CHECK_FOR_INTERRUPTS()`, but are not triggered by other backends sending signals, or are set by other signal handlers (`SIGALRM`, `SIGQUIT`). One idea is to convert those into "procsignals" too, for consistency. In the attached, they can be set (ie by the same backend) with `ProcSignalRaise()`, but it's possible that in future we might have a reason for another backend to set them too, so it seems like a good idea to have a single system, effectively **merging the concepts of "procsignals" and "interrupts"**.

Recap: problems in current system

- Only one latch per process
 - Some exceptions, but it's awkward
 - Can only wait for one latch at a time
- Can't enable / disable specific interrupts
 - `CHECK_FOR_INTERRUPTS()` has some hard-coded exceptions
- Extensions cannot process their own things in `CHECK_FOR_INTERRUPTS()`

Unify "co-operative" interrupt processing and wakeups

In the new system, the Interrupt mechanism covers both:

- You can raise an interrupt for later processing e.g. within a signal handler
- You can raise an interrupt directly in another backend
- You can enable/disable interrupts that will be processed by `CHECK_FOR_INTERRUPTS()`
- You can wait for specific interrupts to arrive
- Configurable "interrupt handler" for each interrupt bit
- Extensions can allocate their own interrupt bits and register handlers for them

Interrupt bits

```
#define INTERRUPT_WAIT_WAKEUP          UINT64_BIT(0)
#define INTERRUPT_TERMINATE             UINT64_BIT(1)
#define INTERRUPT_QUERY_CANCEL          UINT64_BIT(2)
#define INTERRUPT_RECOVERY_CONFLICT     UINT64_BIT(3)
#define INTERRUPT_CONFIG_RELOAD         UINT64_BIT(4)
#define INTERRUPT_LOG_MEMORY_CONTEXT    UINT64_BIT(5)
#define INTERRUPT_BARRIER              UINT64_BIT(6)
#define INTERRUPT_TRANSACTION_TIMEOUT   UINT64_BIT(7)
#define INTERRUPT_IDLE_SESSION_TIMEOUT  UINT64_BIT(8)
#define INTERRUPT_IDLE_IN_TRANSACTION_SESSION_TIMEOUT UINT64_BIT(9)
#define INTERRUPT_CLIENT_CHECK_TIMEOUT  UINT64_BIT(10)
#define INTERRUPT_IDLE_STATS_TIMEOUT    UINT64_BIT(11)
#define INTERRUPT_CLIENT_CONNECTION_LOST UINT64_BIT(12)
#define INTERRUPT_ASYNC_NOTIFY          UINT64_BIT(13)
#define INTERRUPT_SINVAL_CATCHUP        UINT64_BIT(14)
#define INTERRUPT_PARALLEL_MESSAGE       UINT64_BIT(15)
#define INTERRUPT_WALSND_INIT_STOPPING  UINT64_BIT(16)
#define INTERRUPT_WALSND_STOP            UINT64_BIT(17)
#define INTERRUPT_WAL_ARRIVED            UINT64_BIT(18)
#define INTERRUPT_CHECK_PROMOTE          UINT64_BIT(19)
#define INTERRUPT_SUBSCRIPTION_CHANGE    UINT64_BIT(20)
#define INTERRUPT_SHUTDOWN_PARALLEL_APPLY_WORKER UINT64_BIT(21)
#define INTERRUPT_SHUTDOWN_XLOG          UINT64_BIT(22)
#define INTERRUPT_SHUTDOWN_PGARCH        UINT64_BIT(23)
#define INTERRUPT_AUTOVACUUM_WORKER_FINISHED UINT64_BIT(24)

#define BEGIN_ADDIN_INTERRUPTS  25
#define END_ADDIN_INTERRUPTS    64
```

Interrupt handlers

- Each process has a table of *interrupt handlers*
- Similar to signal handlers, but called at CHECK_FOR_INTERRUPTS()
 - You're allowed to do more things than in a signal handler

Standard interrupt handlers

```
/*
 * Set the "standard" set of interrupt handlers, to handle common interrupts
 * that should be handled by all or most child processes.
 *
 * The caller may modify the defaults after calling this with further
 * SetInterruptHandler() or Enable/DisableInterrupt() calls.
 */
void
SetStandardInterruptHandlers(void)
{
    /* All processes should react to barriers and memory context debugging */
    SetInterruptHandler(INTERRUPT_BARRIER, ProcessProcSignalBarrier);
    EnableInterrupt(INTERRUPT_BARRIER);
    SetInterruptHandler(INTERRUPT_LOG_MEMORY_CONTEXT, ProcessLogMemoryContextInterrupt);
    EnableInterrupt(INTERRUPT_LOG_MEMORY_CONTEXT);

    /*
     * Every process should react to INTERRUPT_TERMINATE. But many processes
     * disable this and do their own checks at appropriate times.
     */
    SetInterruptHandler(INTERRUPT_TERMINATE, ProcessTerminateInterrupt);
    EnableInterrupt(INTERRUPT_TERMINATE);

    ...
}
```

Wait loop

Before:

```
for (;;)
{
    ResetLatch(MyLatch);

    if (...)
        break;

    CHECK_FOR_INTERRUPTS();

    (void) WaitLatch(MyLatch,
                    WL_LATCH_SET |
                    WL_EXIT_ON_PM_DEATH,
                    0, WAIT_EVENT_...);
}
```

After:

```
for (;;)
{
    ClearInterrupt(INTERRUPT_WAIT_WAKEUP);
    if (...)
        break;

    CHECK_FOR_INTERRUPTS();

    (void) WaitInterrupt(CheckForInterruptsMask |
                        INTERRUPT_WAIT_WAKEUP,
                        WL_LATCH_SET |
                        WL_EXIT_ON_PM_DEATH,
                        0, WAIT_EVENT_...);
}
```

Wait loop, but ignore query cancel

```
DisableInterrupt(INTERRUPT_QUERY_CANCEL);
for (;;)
{
    CHECK_FOR_INTERRUPTS();

    /* do stuff */

    (void) WaitInterrupt(CheckForInterruptsMask,
                        WL_LATCH_SET |
                        WL_EXIT_ON_PM_DEATH,
                        0, WAIT_EVENT_...);
}

EnableInterrupt(INTERRUPT_QUERY_CANCEL);
```

Wait loop, but do something special on query cancel

```
DisableInterrupt(INTERRUPT_QUERY_CANCEL);
for (;;)
{
    CHECK_FOR_INTERRUPTS();

    if (InterruptPending(INTERRUPT_QUERY_CANCEL))
    {
        elog(NOTICE, "cannot cancel right now");
        ClearInterrupt(INTERRUPT_QUERY_CANCEL);
    }

    (void) WaitInterrupt(CheckForInterruptsMask |
                        INTERRUPT_QUERY_CANCEL,
                        WL_LATCH_SET |
                        WL_EXIT_ON_PM_DEATH,
                        0, WAIT_EVENT_...);
}

EnableInterrupt(INTERRUPT_QUERY_CANCEL);
```

Patch status

- Work in progress
- Feedback appreciated
- Thank you for the feedback so far
 - Thomas Munro
 - Andres Freund
 - Robert haas
 - Joel Jacobson
- Still hoping to commit for version 20

Thank you!

[pgsql-hackers thread](#): "Interrupts vs signals"