Taking advantage of custom bgworkers

PostgreSQL Conference Europe 2013, Dublin, 2013/10/29

Michael Paquier, Member of Technical Staff – PostgreSQL

About your lecturer

- **Michael Paquier**
  - Working on Postgres for VMware
  - Postgres community hacker and blogger
  - Based in Tokyo
- **Contacts:**
  - Twitter: @michaelpq
  - Blog: http://michael.otacoo.com
  - Email: mpaquier@vmware.com, michael@otacoo.com

Agenda

- Introduction to background workers
- Implementation basics
- Good habits and examples
- What next?

Introduction to background workers

Bgworker?

- **Plug-in infrastructure introduced in PostgreSQL 9.3**
- **Child process of postmaster**
  - Similar to a normal backend
  - Control by postmaster, lives and dies with postmaster
  - Signal control centralized under postmaster
  - Assumed to run continuously as a daemon process
- **Run customized code**
  - User-defined code of shared library loaded by PG core server
  - Code loaded at server start
- **Set of APIs for process-related plug-ins**
  - Customizable
  - Extensible
  - Adaptable
  - Dangerous

Features

- **Several options**
  - Access to databases
  - Access to shared memory
  - Serial transactions
- **User-defined parameters**
- **Some control for start/stop/restart of process**
- **Not necessarily visible in pg_stat_***
- **Process listed with suffix bgworker: + $WORKER_NAME as name**

```
$ ps -o pid= -o command= -p `pgrep -f ”worker name"`
$PID postgres: bgworker: worker name
```

Development APIs

- **All in bgworker.h**
- **Main management structure**

```
typedef struct BackgroundWorker
{
        char       bgw_name[BGW_MAXLEN];
        int        bgw_flags;
        BgWorkerStartTime bgw_start_time;
        int        bgw_restart_time;
        bgworker_main_type bgw_main;
        Datum    bgw_main_arg;
} BackgroundWorker;
```

- **Other functions**
  - **RegisterBackgroundWorker**, register bgworker at load phase
  - **BackgroundWorkerBlockSignals/BackgroundWorkerUnblockSignals**
  - **BackgroundWorkerInitializeConnection**, connect to a wanted database
    - Only to catalogs if database name is NULL

Development APIs (2)

- **Flags - bgw_flags**
  - BGWORKER_SHMEM_ACCESS
  - BGWORKER_BACKEND_DATABASE_CONNECTION
- **Start moment – bgw_start**
  - BgWorkerStart_PostmasterStart
  - BgWorkerStart_ConsistentState
  - BgWorkerStart_RecoveryFinished
- **Restart time in seconds - bgw_restart_time**
  - BGW_DEFAULT_RESTART_INTERVAL, 60s by default
  - BGW_NEVER_RESTART
  - Effective for *crashes*
- **Documentation**
  - http://www.postgresql.org/docs/9.3/static/bgworker.html

Implementation basics

“Hello World” class example

- **With most basic implementation**
  - Print once “Hello World”, then exit
  - But this is not funny…
- **Instead => output “Hello World” every 10s to the server logs**

```
LOG:  registering background worker "hello world"
LOG:  loaded library "hello_world"
```

```
$ ps -o pid= -o command= -p `pgrep -f "hello world"`
12642 postgres: bgworker: hello world
```

```
$ tail -n3 pg_log/postgresql-*.log | grep "Hello"
Process: 12642, timestamp: 2013-08-19 12:50:32.159 JSTLOG:  Hello World!
Process: 12642, timestamp: 2013-08-19 12:50:42.169 JSTLOG:  Hello World!
Process: 12642, timestamp: 2013-08-19 12:50:52.172 JSTLOG:  Hello World!
```

Example: Hello World (1)

- **Headers!**

```
/* Minimum set of headers */
#include "postgres.h”
#include "postmaster/bgworker.h”
#include "storage/ipc.h”
#include "storage/latch.h”
#include "storage/proc.h”
#include "fmgr.h”

/* Essential for shared libs! */
PG_MODULE_MAGIC;

/* Entry point of library loading */
void _PG_init(void);

/* Signal handling */
static volatile sig_atomic_t got_sigterm = false;
```

Example: Hello World (2)

- **Initialization with _PG_init()**

```
void
_PG_init(void)
{
    BackgroundWorker worker;
    worker.bgw_flags = BGWORKER_SHMEM_ACCESS;
    worker.bgw_start_time = BgWorkerStart_RecoveryFinished;
    worker.bgw_main = hello_main;
    snprintf(worker.bgw_name, BGW_MAXLEN, "hello world");
    worker.bgw_restart_time = BGW_NEVER_RESTART;
    worker.bgw_main_arg = (Datum) 0;
    RegisterBackgroundWorker(&worker);
}
```

Example: Hello World (3)

- **Main loop**

```
static void
hello_main(Datum main_arg)
{
    pqsignal(SIGTERM, hello_sigterm);
    BackgroundWorkerUnblockSignals();
    while (!got_sigterm)
    {
        int rc;
        rc = WaitLatch(&MyProc->procLatch,
            WL_LATCH_SET | WL_TIMEOUT | WL_POSTMASTER_DEATH,
            10000L);
        ResetLatch(&MyProc->procLatch);
        if (rc & WL_POSTMASTER_DEATH)
            proc_exit(1);
        elog(LOG, "Hello World!"); /* Say Hello to the world */
    }
    proc_exit(0);
}
```

Example: Hello World (4)

- **SIGTERM handler**

```
static void hello_sigterm(SIGNAL_ARGS)
{
    int save_errno = errno;
    got_sigterm = true;
    if (MyProc)
        SetLatch(&MyProc->procLatch);
    errno = save_errno;
}
```

- **Makefile**

```
MODULES = hello_world
PG_CONFIG = pg_config
PGXS := $(shell $(PG_CONFIG) –pgxs)
include $(PGXS)
```

Example: Hello World – Conclusion

- **Good things**
  - Postmaster death correctly managed
  - Management of SIGTERM
  - Use of a Latch
- **And not-so-good things**
  - Avoid shared memory connection if possible
    - Might lead to memory corruption
    - Use a private latch
  - Avoid database connection if not necessary
  - Management of SIGHUP

Just don’t forget that (1)

- **Consistency with existing backend code**
  - Don’t reinvent the wheel!
- **Reload parameters**
  - Handling of SIGHUP and ProcessConfigFile important!
  - Postmaster sends signal to workers, but workers should handle it properly
- **Test your code before putting it in production, especially if…**
  - bgworker interacts with the OS/database
  - Access to shared memory used
- **Security**
  - That’s C!
  - Door to security holes
    - Ports opened on a bgworker
    - Interactions with other components
  - Easy to break server…

Just don’t forget that (2)

- **Use a private latch if possible**
- **Limit access to shared memory**
  - Flag BGWORKER_SHMEM_ACCESS
  - Don’t play with security
- **Limit access to database**
  - Flag BGWORKER_SHMEM_ACCESS | BGWORKER_BACKEND_DATABASE_CONNECTION
- **Do NOT use pg_usleep, does not stop at postmaster death**
- **Load with _PG_init() and PG_MODULE_MAGIC to enable it!**
- **Headers necessary to survive**

```
#include "postgres.h”
#include "postmaster/bgworker.h”
#include "storage/ipc.h”
#include "fmgr.h”
```

Just don't forget that (3)

- **No physical limit of bgworkers allowed**
  - MaxBackends calculated from number of registered workers
  - Lot of bgworkers = risk of OOM on standby
  - Be sure to not have an extravagant number of workers
  - Fixed in 9.4~ with max_worker_processes
- **Code loading**
  - Set shared_preload_libraries in postgresql.conf
  - Entry point is _PG_init()
  - Register your worker
- **Set signal functions, then unblock signals**

```
pqsignal(SIGHUP, my_worker_sighup);
pqsignal(SIGTERM, my_worker_sigterm);
BackgroundWorkerUnblockSignals();
```

Just don’t forget that (4)

- **One last thing… Limitations for one-time tasks**
  - Workers designed to always restart, like daemons
  - Possible to combine NEVER_RESTART with exit code != 0 for definite stop, not that intuitive
  - Cannot start workers at will, always at server startup
  - When stopped like that, can never be restarted

Good habits and examples

What should do a bgworker?

- **Like a daemon process**
  - Interact with external components for an interval of time
  - Monitor activity inside and outside server
  - Check slave status (trigger an email if late on replay?)
- **Like a Postgres backend**
  - Run transactions, queries and interact with databases
  - Receive, proceed signal
  - Proceed signals
  - Use existing infrastructure of server
  - Run statistics
  - Other things not listed here

Custom parameters

- **Loaded in _PG_init**
- **Advice for name convention**
  - $WORKER_NAME.$PARAMETER_NAME
  - Not mandatory though… Feel free to mess up everything
- **Separate config file?**
- **Same control granularity as server**
  - APIs in guc.h
  - Int, float, bool, enum, string
  - Type: sighup, postmaster

```
void DefineCustomIntVariable(
        const char *name,
        const char *short_desc,
        const char *long_desc,
        int *valueAddr,
        int bootValue,
        int minValue,
        int maxValue,
        GucContext context,
        int flags,
        GucIntCheckHook check_hook,
        GucIntAssignHook assign_hook,
        GucShowHook show_hook);
```

Timestamps

- **Timestamps in transactions**
  - Set in postgres.c, not in worker code!
  - Calls to SetCurrentStatementStartTimestamp()
    - *Before* transaction start
    - And *Before* extra query execution

```
/* Start transaction */
SetCurrentStatementStartTimestamp()
StartTransactionCommand();

/* Run queries (not necessary for 1st one in transaction) */
SetCurrentStatementStartTimestamp()
[… Run queries …]
```

Statistics

- **Mainly calls to pgstat_report_activity**
  - STATE_RUNNING with query string before running query
  - STATE_IDLE when transaction commits
  - Activity mainly reported in pg_stat_activity
- **Advantage of reporting stats**
  - Good for maintenance processes
  - Check if process is not stuck
  - For database processes only
- **When not necessary?**
  - Utility workers (no direct interaction with server)
  - Cloud apps, users have no idea of what is running for them here
- **APIs of pgstat.h**

Transaction flow

- **All the APIs of xact.c**

```
/* Start transaction */
SetCurrentStatementStartTimestamp()
StartTransactionCommand();
SPI_connect();
PushActiveSnapshot(GetTransactionSnapshot());

/* Run queries */
SetCurrentStatementStartTimestamp()
pgstat_report_activity(STATE_RUNNING, $QUERY)
[… Run queries …]

/* Finish */
SPI_finish();
PopActiveSnapshot();
CommitTransactionCommand();
pgstat_report_activity(STATE_IDLE, NULL);
```

Execute queries (1)

- **With SPI, common facility of all Postgres modules and core**
- **Functions in executor/spi.h**
  - SPI_connect to initialize facility
  - SPI_finish to clean up
  - SPI_execute to run query
  - SPI_getbinval to fetch tuple values
- **Prepared queries**
  - SPI_prepare to prepare a query
  - SPI_execute_plan to execute this plan
  - etc.
- **Use and abuse of StringInfo and StringInfoData for query strings ☺**

Execute queries (2)

- **Common way of fetching tuple results**

```
/* Execute query */
ret = SPI_execute("SELECT intval, strval FROM table",
                                true, 0);
if (ret != SPI_OK_SELECT)
      elog(FATAL, "Fatal hit…");
/* Fetch data */
for (i = 0; i < SPI_processed; i++)
{
      intValue = DatumGetInt32(
                  SPI_getbinval(SPI_tuptable->vals[i],
                  SPI_tuptable->tupdesc,
                  1, &isnull));
      strValue = DatumGetCString(
                  SPI_getbinval(SPI_tuptable->vals[i],
                  SPI_tuptable->tupdesc,
                  2, &isnull));
}
```

Example - kill automatically idle connections

- **Use of the following things**
  - Custom parameters
  - Timestamps
  - Transaction
  - SPI calls
- **Query used by worker process**

```
SELECT pid, datname, usename,
                pg_terminate_backend(pid) as status
FROM pg_stat_activity
WHERE now() - state_change > interval '$INTERVAL s' AND
      pid != pg_backend_pid();
```

- **Interval customizable with parameter**
  - Name: kill_idle.max_idle_time
  - Default: 5s, Max value: 1h

Next example, cut automatically idle connections

- **Worker process**

```
$ ps -o pid= -o command= -p `pgrep -f "kill_idle"`
23460 postgres: bgworker: kill_idle
```

- **Disconnection activity in logs**

```
$ tail -n 2 postgresql-*.log | grep Disconnected
LOG:  Disconnected idle connection: PID 23564 mpaquier/mpaquier/none
LOG:  Disconnected idle connection: PID 23584 postgres/mpaquier/none
```

- **Statistic activity**

```
postgres=# SELECT datname, usename, substring(query, 1, 38)
                  FROM pg_stat_activity WHERE pid = 23460;
 datname  | usename  |              substring
----------+----------+---------------------------------------
 postgres | mpaquier | SELECT pid, pg_terminate_backend(pid)
(1 row)
```

More material?

- **Documentation**
  - http://www.postgresql.org/docs/9.3/static/bgworker.html
- **Bgworker modules popping around**
  - Mongres:
    - Get MongoDB queries and pass them to Postgres
    - https://github.com/umitanuki/mongres
  - contrib/worker_spi
    - All the basics in one module
    - 9.4~ stuff also included on master
  - A future pg_cron?
  - Examples of today and more => pg_workers
    - https://github.com/michaelpq/pg_workers
    - kill_idle https://github.com/michaelpq/pg_workers/tree/master/kill_idle
    - hello_world https://github.com/michaelpq/pg_workers/tree/master/hello_world
    - Under PostgreSQL license

What next?

Bgworkers, and now?

- **With stable 9.3 APIs, wide adoption expected**
- **Many possibilities**
  - Statistics-related processes
  - Maintenance, cron tasks
    - Reindex automatically invalid indexes
    - Kill inactive connections after certain duration (pg_stat_activity + pg_terminate_backend) combo
- **HA agents, Pacemaker, Corosync, watchdogs**
- **Health checker**
  - Disk control: Stop server if free disk space <= X%
  - Automatic update of parameter depending on environment (cloud-related)
- **License checker: Ideal for Postgres server controlled in cloud?**

Bgworkers, and in core?

- **Dynamic bgworkers – new sets of APIs in 9.4~**
  - Infrastructure for parallel query processing
  - Backward compatible with 9.3
  - Start/stop/restart at will
    - Main worker function loaded externally
    - No need of static loading
    - Not adapted for daemon processes
  - Dynamic area of shared memory for communication between backends
    - Parallel sequential scan
    - Parallel sort
    - Transaction snapshots

Thanks!
Questions?