Deep dive into Resource Manager, Error and Interrupt Handling

Rahila Syed, PostgreSQL developer

Agenda

- Resource Manager – Definition and properties
- Resource Owner
- Resource Type
- Resource Manager – Example
- Error Handling
- Main Loop error handler
- Critical Section
- Interrupt Handling
- Things to Note

Resource Manager - Definition

- Tracks and manages life-cycle of resources
- Mechanism to ensure resources are freed at the right time
- ResourceOwner - Object to track resources
- Ex. TopTransaction, SubTransaction, Portal
- Resources - Assets tracked by ResourceOwner
- Ex. Buffers, locks, cache

Resource Manager - Properties

- Hierarchy: Resource owners form a tree structure, Ex. transaction-level owners are parents and portal-level owners are children
- Reference Counting: Some resource type maintains reference counts through the resource manager
- Automatic Cleanup: When a resource owner is released, all resources it and its children own are automatically cleaned up

Resource Owner - Basic operations

- Create a ResourceOwner
- Associate or dissociate some resource with ResourceOwner
- Release a ResourceOwner’s assets
- Delete a ResourceOwner

Resource Type

- Each resource type has:
- A name for identification
- A release phase (BEFORE_LOCKS, LOCKS, AFTER_LOCKS)
- A priority within that phase
- Callback functions for release and debugging

Three phase release-model

- BEFORE_LOCKS: Release non-lock resources
- LOCKS: Release lock resources
- AFTER_LOCKS: Final cleanup

Resource Owner - Usage

- ResourceOwnerCreate(parent, name)
- ResourceOwnerRemember(resowner, res, restype)
- ResourceOwnerForget(resowner, res, restype)
- ResourceOwnerRelease(resowner, phase, isCommit, isTopLevel)
- ResourceOwnerDelete(resowner)

Transaction’s resources

```
CREATE TEMP TABLE top_transaction_data (
    id INTEGER,
    value TEXT,
   created_at TIMESTAMP DEFAULT NOW()
)
```

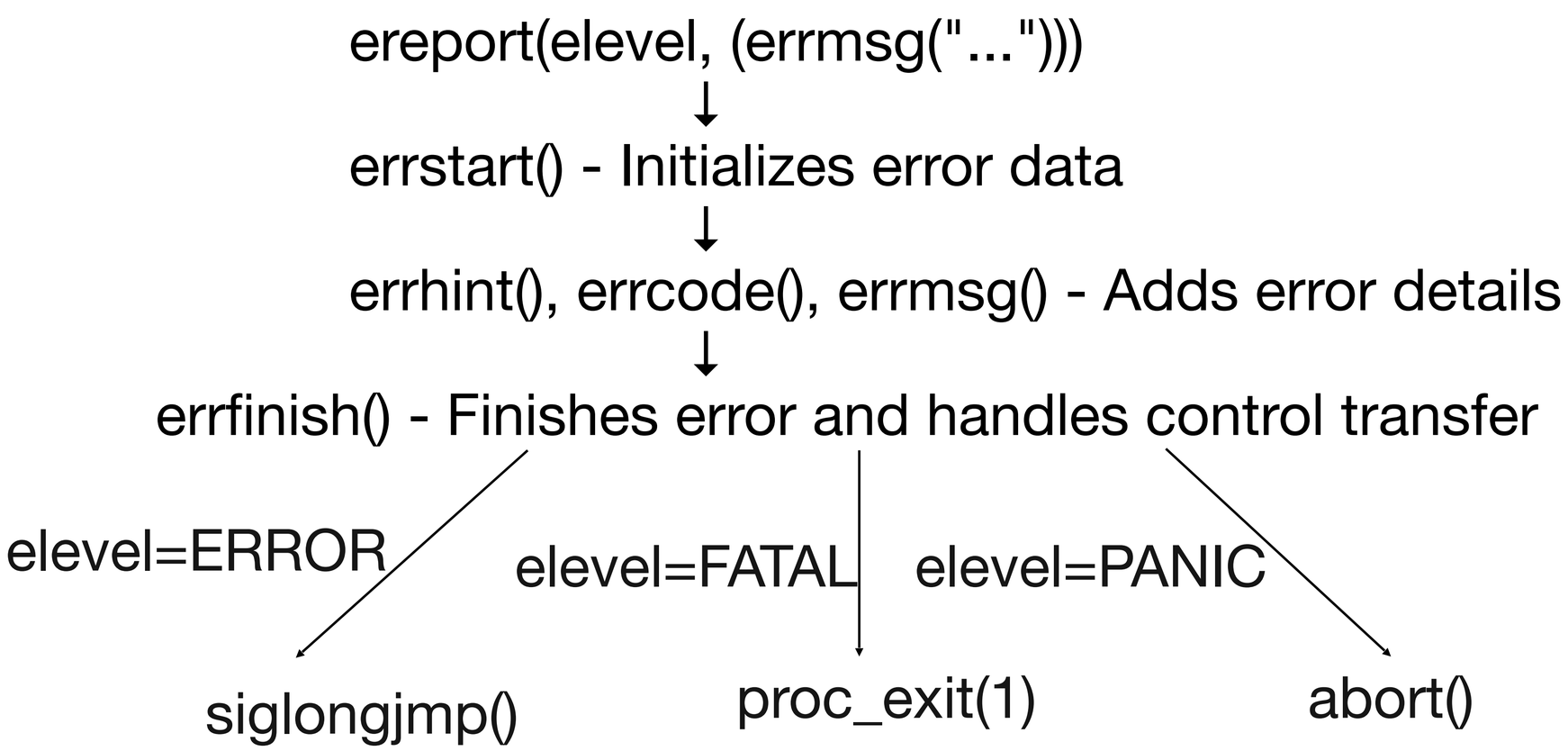
```
------ ResourceOwnerCreate (parent = NULL, name = "TopTransaction")
------ ResourceOwnerCreate (parent = "TopTransaction", name = "Portal")
------  ResourceOwnerRemember (owner = Portal, resource = Relation, resource_type = &relref_resowner_desc)
              .
              .
              .
------ ResourceOwnerForget ( owner = Portal, resource = Relation, resource_type = &relref_resowner_dec)
              .
              .
              .
------ ResourceOwnerRelease (owner = "Portal", phase = RESOURCE_RELEASE_BEFORE_LOCKS)
------ ResourceOwnerRelease (owner = "Portal", phase = RESOURCE_RELEASE_LOCKS)
------ ResourceOwnerRelease (owner = "Portal", phase = RESOURCE_RELEASE_AFTER_LOCKS)
------ ResourceOwnerDelete (owner = "Portal")
------ ResourceOwnerRelease (owner = "TopTransaction", phase = RESOURCE_RELEASE_BEFORE_LOCKS)
------ ResourceOwnerRelease (owner = "TopTransaction", phase = RESOURCE_RELEASE_LOCKS)
------ ResourceOwnerRelease (owner = "TopTransaction", phase = RESOURCE_RELEASE_AFTER_LOCKS)
------ ResourceOwnerDelete (owner = "TopTransaction")
```

Error Handling

PostgreSQL defines three error levels that affect system behavior:

- ERROR: Aborts current transaction
- FATAL: Terminates current session
- PANIC: System-wide reset, all sessions terminated

Control flow



Control Transfer

```
postgres=# insert into tab values(1);
ERROR:  relation "tab" does not exist
LINE 1: insert into tab values(1);
                    ^
```

postgres.c

```
4481         */
4482
4483        if (sigsetjmp(local_sigjmp_buf, 1) != 0)
4484        {
4485                /*
4486                 * NOTE: if you are tempted to add more code in this if-block,
4487                 * consider the high probability that it should be in
4488                 * AbortTransaction() instead.  The only stuff done directly here
4489                 * should be stuff that is guaranteed to apply *only* for outer-level
4490                 * error recovery, such as adjusting the FE/BE protocol status.
4491                 */
4492
4493                /* Since not using PG_TRY, must reset error stack by hand */
> 4494                error_context_stack = NULL;
4495
4496                /* Prevent interrupts while cleaning up */
4497                HOLD_INTERRUPTS();
```

```
multi-thre Thread 0x7f4d493da7 (src) In: PostgresMain                    L4494 PC:
(gdb) n
(gdb) n
(gdb) n
0x00005ce53d2d58e4 in PostgresMain (dbname=0x5ce5602e8630 "postgres", username=0x5ce5602e8620 "rahila")
    at postgres.c:4483
(gdb) c
```

Main loop error handler

- Top-level sigsetjmp cleanup depends on the type of the process – auxiliary process, background worker or client backend.
- Operations like set error_context_stack = NULL, HOLD_INTERRUPTS() , EmitErrorReport() are run for all processes
- Processes that run transactions, do most of the cleanup via AbortTransaction.

Unifying Error Handling

- A unified handler function with following sections
  - Common cleanup
  - Process-specific cleanup
  - Flags to indicate which subsystem needs reset
  - Run AbortTransaction if needed
  - Exit if can_continue = false

START_CRIT_SECTION/END_CRIT_SECTION

- Any ERROR or FATAL inside a critical section is automatically escalated to PANIC
- Inside critical sections, the system is in a state where partial failure is unacceptable.
- Ex.
  - WAL Record Writing: Must complete entirely
  - Buffer Management: Page state modifications must be atomic from a recovery perspective
  - Catalog Updates: Multi-step updates affecting system consistency

Interrupt Handling

- Provides safe points where interrupts can be processed
- Signal handlers set interrupt-pending flags without handling them immediately
- CHECK_FOR_INTERRUPTS() is called strategically at points where system state is self-consistent
- Ex. Every iteration of main loop in a long running process, before I/O operations, between statement executions

Processing Interrupts

- Check the pending interrupt flags
- Clear the pending interrupt flags
- Determine interrupt type
- Execute appropriate handler

Things to note

- Creating a resource from interrupt handlers can lead to warnings like ResOwnerEnlarge called after release started
- No Resource Owner means that the resources are released during process exit or explicitly
- LWLockReleaseAll() should be done early on while executing the error recovery or process exit.

THANK YOU!