Threading in Firebird and the Future

Ann W. Harrison
James A. Starkey

A Word of Thanks to our Sponsors

MOSCOW EXCHANGE
Platinum

SITA SOFTWARE
Platinum

IBPhoenix
Platinum

IBSurgeon
Platinum

REDSOFT

IBObjects

CopyCat

Upscene

Why Threads?

Improve multi-user performance

Utilize multi-processor systems

Databases are too easy to multi-thread

What are Atomic Instructions?

Most machine instructions can be interrupted, allowing the world to change.

Atomic instructions (e.g. CAS) run to completion.

Essential for multi-thread performance.

Wasn’t Firebird Always Threaded?

Earliest versions of shared server ran query for one user until it stalled before responding to next user.

Not friendly.

Multi-threaded server runs to the next wait or for a fixed period plus the time to make the database state consistent.

Threads never run concurrently.

Firebird Classic

Designed for VAX Clusters

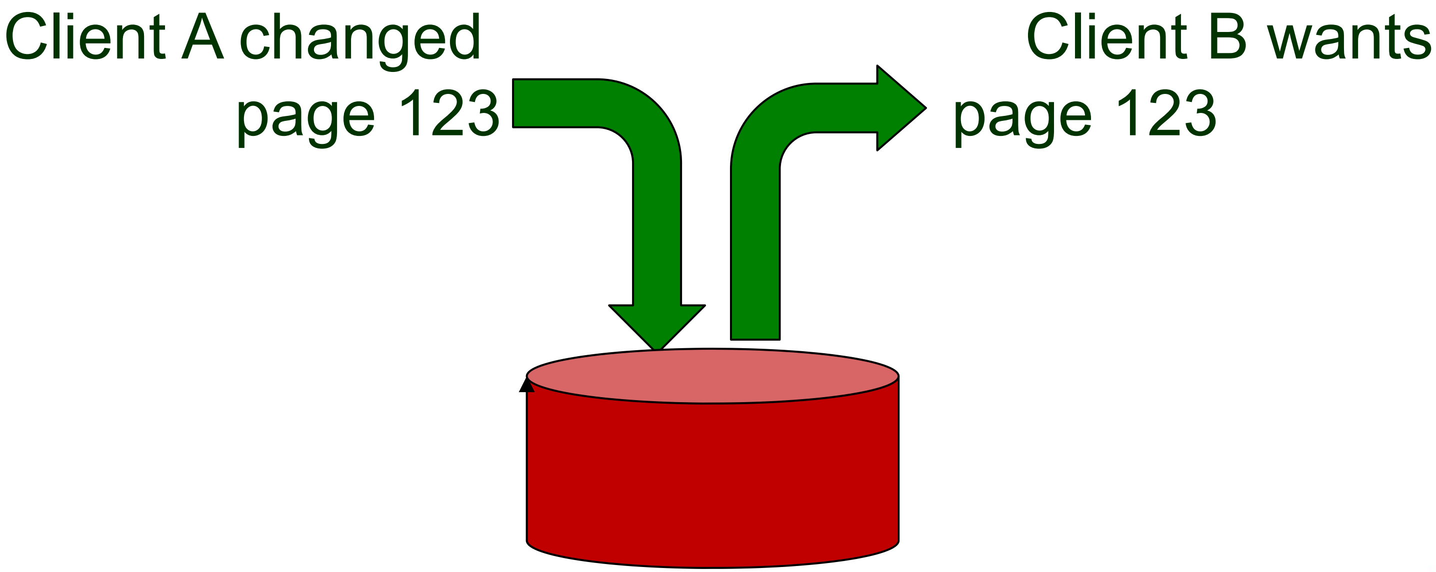
- Multiple independent computers
- Shared intelligent disk controller
- Cluster-wide lock manager

Firebird Classic Multi-Processor

- Single Machine, Multi-Processor
- O/S schedules Firebird clients on processors
- Clients share
  - Disk
  - Lock manager
- Clients do not share
  - Page Cache
  - Metadata

Non-shared cache

Firebird classic, super classic



Yes, that is really a disk write

Shared cache - Superserver

Client A changed page 123

1

Client B wants page 123

2

Client A releases lock on page 123

3

Client B locks page 123

Threading, 101

Thread

- PC: Instruction stream of control
- Dedicated Stack (1 mb+)
- Thread specific data
- All threads share process memory
- Expensive to create, cheap to use
- (If you don’t thrash)

Threading 101

Interlocked Instruction: Atomic compare and swap

- Compares given value to value at given address
- If equal, store new value at given address
- If not, fails and does nothing

Interlocked instructions are the mortar of multi-threading

Threading 101

Non-interlocked data structures

- Data structures managed only by interlocked instructions
- Completely non-blocking
- The fastest – and hardest – form of multi-programming

Threading 101

RW-lock, aka SyncObject

- Can be locked for read/shared
- Can be locked for write/exclusive
- Blocks until access can be granted
- Monitor semantics: Thread doesn’t lock against itself
- Implemented with interlocked CAS

Threading 101

Coarse grain multi-threading

- Single mutex controls an entire subsystem
- Individual data structures are not interlocked

Fine grain multi-threading

- Individual RW-lock per data structure
- Allows many threads to share a subsystem

Threading 101

- Dedicated Thread
  - Thread assigned specific task
  - Garbage collector, network listener, etc.
- Client thread
  - Thread executing user request
- Worker Thread
  - Thread idle or executing user request
- Thread pool
  - Manages worker threads

Threading Models

Thread per connection

- Worker thread assigned at connection time
- Worker thread == Client thread
- Idle client threads consume resources
- Many connections => high contention

Threading Models

Limited worker threads

- Limit active worker threads to approx. number of processors
- User requests queued until work thread becomes available
- If worker thread stalls (page read), thread pool can release next user request
- Utilizes processors without unnecessary contention

Threading Models

Limited Worker Threads:

- Dedicated listener thread waits for readable socket
- Connection object (on listener thread) does socket read
- When packet is complete, connection object queue to thread pool
- When worker thread becomes available, connection object is executed

Threading Model

Thread per connection is first step

Limited worker threads is essential for scalability

Interbase Threads: The Beginning

The concept of threads was known at the birth of Interbase, but no implementations existed on small machines.

SMP didn't exist in the mini or workstation world

The initial version of Interbase used signals for communication

User requests executed with "looper"; when a request stalled, another request could run

Interbase Theads: The V3 Disaster

Apollo was the first workstation vendor with threads

I implemented a VMS threading package

Sun’s first attempt at threads didn’t even compile

Interbase V3 was going to be mixed signals + threads

Then disaster: Apollo Domain had unfixable architectural flaw mixing threads and signals

A **long** slip ensued

Interbase Threads: V3 Reborn

The engine was either threaded or signal based

Dedicated threads for lock manager, event manager, etc.

Server was thread per client

Engine continued with coarse grain multi-threading

Firebird Threading: Vulcan

- Vulcan, now deceased, introduced limited fine grain multi-threading
- Threads synchronized with SyncObject: User mode read/write locks with monitor semantics
- SMP had arrived, followed shortly by processor based threads

Some Performance Lessons

The goal is to saturate CPU, network, memory, and disk bandwidth simultaneously.

There is no reason to run more worker threads than cores (and many reasons not to), but

A stalled thread is an efficient way to maintain request state (death to “looper”!)

A Winning Architecture

A single dedicated thread waiting for readable sockets

Request starts are posted to thread manager for available worker thread

When active worker threads drops below threshold, a pending request is assigned a worker thread

A stalling thread checks in with thread manager to drop the number of active worker threads

An unstalled request bumps the number of a.w.t.