

# Samba locking architecture

SNIA SDC 2020

Volker Lendecke

Samba Team / SerNet

Sept 2020

# Implementing SMB on top of Posix file APIs

- ▶ SMB and Posix both define file access methods
  - ▶ SMB is a protocol, Posix defines a local API
  - ▶ Protocols are understood as “API” these days as well
  - ▶ Both define files, directories and their metadata
- ▶ Superficially similar: You can open/close/read/write, you can create, list and delete files
- ▶ SMB and Posix differ a lot in the details
  - ▶ Case sensitivity
  - ▶ Semantics to delete a file
  - ▶ SMB has FileChangeNotify, Posix does not
  - ▶ Locking semantics are vastly different: share modes
  - ▶ SMB as a protocol defines cache coherency with leases
- ▶ Samba implements SMB semantics on top of Posix files

# Samba architecture

- ▶ For every client Samba forks a new process
- ▶ Distinct memory spaces in every process
- ▶ MS-SMB2 and MS-FSA suggest a lot of shared tables
  - ▶ Lists of clients, tree connects, open files and others
- ▶ Samba can't use any of those data structures directly
- ▶ Samba stores SMB2 and FSA tables in file-backed Key/Value stores
- ▶ The lowest layer is tdb (<https://tdb.samba.org>)
  - ▶ This is a simple database API. It was inspired by the realisation that in Samba we have several ad-hoc bits of code that essentially implement small databases for sharing structures between parts of Samba.
- ▶ Shared hash table with hash-chain locking

# Trivial Database tdb

- ▶ Long ago Samba used System V shared memory
  - ▶ Data protection using System V semaphores
  - ▶ The world settled on mmap and fcntl
- ▶ TDB is a library for a shared hash table backed by mmap
- ▶ `tdb_fetch()/tdb_store()/tdb_delete()` for data access
- ▶ Hash chain locking available for API users: `tdb_chainlock()`
- ▶ Many optimizations and extensions over the years
  - ▶ Transactions using `fsync()/msync()`
  - ▶ Many small and large performance improvements
- ▶ Still 32-bit: Good enough for transient locking data
  - ▶ AD Controller switched to OpenLDAP's `lmdb`

# Clustering tdb

- ▶ tdb uses a local shared memory segment for performance
  - ▶ Shared memory not remotely accessible
- ▶ Add a dbwrap layer around tdb that covers required usecases
- ▶ `dbwrap_fetch_locked()` locks a record and gives r/w access to its contents
  - ▶ All write and delete access needs to happen under such a lock
- ▶ `dbwrap_parse_record()` allows unlocked read
- ▶ Flexible implementations for dbwrap:
  - ▶ Default is local tdb
  - ▶ `dbwrap_file` implemented one file per record in a cluster file system, proved initial scalability
  - ▶ `dbwrap_ctdb` implements API through clustered tdb

## dbwrap API excerpt

```
struct db_record *dbwrap_fetch_locked(  
    struct db_context *db,  
    TALLOC_CTX *mem_ctx,  
    TDB_DATA key);  
TDB_DATA dbwrap_record_get_value(  
    const struct db_record *rec);  
NTSTATUS dbwrap_record_store(  
    struct db_record *rec,  
    TDB_DATA data,  
    int flags);  
NTSTATUS dbwrap_record_delete(  
    struct db_record *rec);
```

# Using dbwrap records

- ▶ `dbwrap_fetch_locked` represents both record data and the lock
- ▶ `dbwrap_record` is `talloc`-based, `talloc_free()` unlocks the record
- ▶ The only way to access and modify the record data is via the functions acting on `struct db_record`
- ▶ `dbwrap_fetch_locked` is implemented as a function pointer inside `struct db_context`
  - ▶ Likewise, `dbwrap_record_delete` is a function pointer inside `db_record`
- ▶ Other K/V store implementations can implement those “methods”

# Opening a file that has a conflicting lease

- ▶ For cache coherency, SMB implements leases
- ▶ A lease is a guarantee that the lease holder exclusively has a file open
  - ▶ This allows extensive caching of reads and writes
  - ▶ There's more than one type of lease, but for this talk an exclusive lease should be sufficient
- ▶ Client A opens a file, gets a lease. smbd "a" updates locking.tdb
- ▶ Client B wants to open a file, smbd "b" finds the lease being taken
- ▶ "b" finds "a" in locking.tdb, asks "a" to give up the lease
- ▶ "a" asks its Client to give up the lease and modifies locking.tdb
- ▶ Questions:
  - ▶ How does "a" tell "b" about the lease state change?
  - ▶ What does "b" do in the meantime?

# Watching dbwrap records

- ▶ `smbd` “a” gets a message from “b” to break a lease
- ▶ “a” could record the source of this message for a later reply
- ▶ What if multiple clients want to break this lease simultaneously?
  - ▶ Maintaining list of recipients is tedious and error-prone
- ▶ In a previous implementation, the lease breakers added their PID (“b” in this case) to the `locking.tdb` file metadata
- ▶ A lease break will eventually manifest by a changed `locking.tdb` record
- ▶ `dbwrap_watched_watch_send()` abstracts waiting for record changes

## API excerpt for watching records

```
struct tevent_req *dbwrap_watched_watch_send(  
    TALLOC_CTX *mem_ctx,  
    struct tevent_context *ev,  
    struct db_record *rec,  
    struct server_id blocker);
```

- ▶ Watching a record creates an asynchronous computation
  - ▶ More fancy languages than C would call `tevent_req` a promise
  - ▶ The `tevent_req` gets fulfilled when “rec” changes
  - ▶ `tevent` is LGPL, usable outside of Samba and makes async programming in C a lot of fun
- ▶ “blocker” is a PID that holds the current resource, i.e. “a”
- ▶ The `tevent_req` also fires when the blocker dies

# Implementation of dbwrap\_watch

- ▶ Layered on top of any dbwrap implementation:

```
struct db_context *db_open_watched(  
    TALLOC_CTX *mem_ctx,  
    struct db_context **backend,  
    struct messaging_context *msg);
```

- ▶ This enables watching records on any lower-level K/V store
- ▶ When using such a watched db, transparently a list of watchers is added to each record
- ▶ API users still call dbwrap\_fetch\_locked() & friends
- ▶ dbwrap\_record\_store() on a watched record will ping all watchers

# Answering oplock questions

- ▶ When asking for a lease break, the breaking process “b” watches the locking.tdb record when it finds a lease being granted, so:
- ▶ How does “a” tell “b” about the lease state change?
  - ▶ The lease break triggers a record\_store by “a” on the locking.tdb record
  - ▶ dbwrap\_watch takes care of informing “b”
- ▶ What does “b” do in the meantime?
  - ▶ It serves SMB while dbwrap\_watched\_watch\_send()'s promise gets fulfilled

# Fixing tdb over-locking

- ▶ tdb at its core is a shared hash table
- ▶ `dbwrap_fetch_locked()` locks more than it should: It uses `tdb_chainlock()` locking a whole hash chain
- ▶ `smbd` has to do expensive operations under a lock:
  - ▶ `open()`, `unlink()` and even `close()` can take ages
  - ▶ Networked and clustered file systems can be very slow
- ▶ Clustered Samba has a problem:
  - ▶ `smbd` wants to open a file, takes a chainlock
  - ▶ File system goes to lunch, `smbd` blocks in open while holding the lock
  - ▶ `ctdb` finds out the node is in trouble, needs to expel the node
  - ▶ Expelling a node means a recovery walking the whole tdb
  - ▶ tdb is partially blocked, i.e. `ctdb` can't cleanly expel the node

# Implementing per-record locks, a bit of history

- ▶ Samba depends upon crash-resilient, persistent tdb files:
  - ▶ Workstation password maintained in secrets.tdb
  - ▶ Configuration in registry.tdb
- ▶ tdb transactions: Lock everything, do the transaction, unlock
- ▶ This needs to be extended to the cluster, however:
  - ▶ ctdb has no global state, can't do a global lock
- ▶ Samba implements advisory locks on top of transient g\_lock.tdb
- ▶ A global lock is represented by one record
  - ▶ Lock holders enter themselves into a record
  - ▶ Conflicting lockers patiently wait using dbwrap\_watch
- ▶ tdb chain locks only held very briefly
  - ▶ No blocking operations under the tdb chainlock
- ▶ With g\_lock.tdb, ctdb transactions are now safe

# Implementing per-record locks

- ▶ Requirement: Prevent Samba version mismatch in a cluster
  - ▶ For this, `g_lock` was extended to carry the version string
  - ▶ `g_lock_write_data` and `g_lock_parse` access the payload
- ▶ First `smbd` takes a shared lock on a version record
- ▶ Try to upgrade to an exclusive lock
  - ▶ If that works, we're the first one: Write our version
  - ▶ If that fails, someone else exists in the cluster, compare the version
- ▶ Downgrade to a shared lock for others to join
- ▶ Implementing `locking.tdb` on top of `g_lock` now solves the `tdb` overlocking problem
  - ▶ `tdb chainlocks` not taken while `smbd` sits in `unlink()`

# locking.tdb payload

uint32 share\_mode\_data\_len

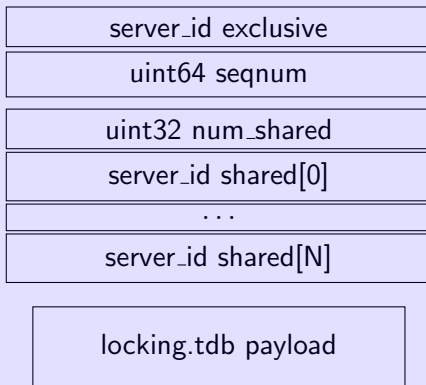
NDR share\_mode\_data

share\_mode\_entry[0]

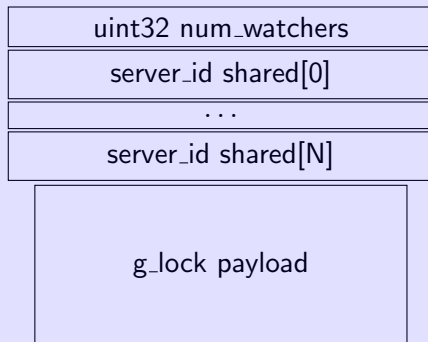
...

share\_mode\_entry[N]

## ...wrapped in g\_lock data



...wrapped in dbwrap\_watch data



# Wrapping up

- ▶ Samba implements SMB file semantics using shared memory
  - ▶ The dbwrap abstraction extends this to a cluster
- ▶ Asynchronous monitoring of records is done by an abstract API
  - ▶ `dbwrap_watch` hides the complexity of explicitly waiting for leases to be broken
- ▶ Per-record locks are implemented on top of `dbwrap_watch`
- ▶ Implementation efficiency by avoiding `memcpy` wherever possible
- ▶ This hierarchy could be split at any layer for other K/V stores
  - ▶ `g_lock` utilizes simple primitives
  - ▶ K/V stores without locks could utilize `g_lock` for alternatives to `ctdb`

vl@samba.org / vl@sernet.de  
<https://www.samba.org> / <https://www.sernet.de/>