NAME

lock, unlock, query_lock, set_lock_cache_enable, lock_cache_enabled − Class ss_m Methods for Locking

SYNOPSIS

```
#include <sm_vas.h>  // which includes sm.h

static rc_t                 lock(
    const lvid_t&               lvid,
    lock_mode_t                 mode,
    lock_duration_t             duration = t_long,
    long                        timeout = WAIT_SPECIFIED_BY_XCT);


static rc_t                 lock(
    const lockid_t&             lockid,
    lock_mode_t                 mode,
    lock_duration_t             duration = t_long,
    long                        timeout = WAIT_SPECIFIED_BY_XCT);

static rc_t                 unlock(const lockid_t& lockid);

static rc_t                 query_lock(
    const lockid_t&             lockid,
    lock_mode_t&                mode,
    bool                        implicit = false);

static rc_t                 set_lock_cache_enable(bool enable);
static rc_t                 lock_cache_enabled(bool& enabled);

static rc_t                 set_escalation_thresholds(
    int4                        toPage,
    int4                        toStore,
    int4                        toVolume);

static rc_t                 get_escalation_thresholds(
    int4&                       toPage,
    int4&                       toStore,
    int4&                       toVolume);

static rc_t                 dont_escalate(
    const lockid_t&             n,
    bool                        passOnToDescendants = true);
static rc_t                 dont_escalate(
    const lvid_t&               lvid,
    bool                        passOnToDescendants = true);

// Lock ID

class lockid_t {
public:
    //
    // The lock graph consists of 6 node: volumes, stores, pages, key values,
    // records, and extents. The first 5 of these form a tree of 4 levels.
    // The node for extents is not connected to the rest.
    // The node_space_t enumerator maps node types to integers.
    // These numbers are used for
    // indexing into arrays containing node type specific info per entry (e.g
    // the lock caches for volumes, stores, and pages).
    //
    enum name_space_t {
    t_bad        = 10,
    t_vol        = 0,
    t_store        = 1,    // parent is 1/2 = 0 t_vol
    t_page        = 2,    // parent is 2/2 = 1 t_store
    t_kvl        = 3,    // parent is 3/2 = 1 t_store
    t_record    = 4,    // parent is 4/2 = 2 t_page
    t_extent    = 5,
    t_user1        = 6,
    t_user2        = 7,    // parent is t_user1
    t_user3        = 8,    // parent is t_user2
    t_user4        = 9    // parent is t_user3
    };

    struct user1_t  {
    uint2_t        u1;
            user1_t() : u1(0)  {}
            user1_t(uint2_t v1) : u1(v1)  {}
    };

    struct user2_t : public user1_t  {
    uint4_t        u2;
            user2_t() : u2(0)  {}
            user2_t(uint2_t v1, uint4_t v2): user1_t(v1), u2(v2)  {}
    };

    struct user3_t : public user2_t  {
    uint4_t        u3;
            user3_t() : u3(0)  {}
            user3_t(uint2_t v1, uint4_t v2, uint4_t v3)
                : user2_t(v1, v2), u3(v3)  {}
    };

    struct user4_t : public user3_t  {
    uint4_t        u4;
            user4_t() : u4(0)  {}
            user4_t(uint2_t v1, uint4_t v2, uint4_t v3, uint4_t v4)
                : user3_t(v1, v2, v3), u4(v4)  {}
    };

    bool operator==(const lockid_t& p) const;
    bool operator!=(const lockid_t& p) const;
    friend ostream& operator<<(ostream& o, const lockid_t& i);

    uint4_t             hash() const;
    void             zero();

    name_space_t         lspace() const;
```

```
        vid_t             vid() const;
        const snum_t&         store() const;
        const extnum_t&         extent() const;
        const shpid_t&         page() const;
        const slotid_t&         slot() const;
        uint2_t            u1() const;
        uint4_t            u2() const;
        uint4_t            u3() const;
        uint4_t            u4() const;

        void            set_ext_has_page_alloc(bool value);
        bool            ext_has_page_alloc() const ;

        NORET            lockid_t() ;
        NORET            lockid_t(const vid_t& vid);
        NORET            lockid_t(const extid_t& extid);
        NORET            lockid_t(const stid_t& stid);
        NORET            lockid_t(const lpid_t& lpid);
        NORET            lockid_t(const stpgid_t& stpgid);
        NORET            lockid_t(const rid_t& rid);
        NORET            lockid_t(const kvl_t& kvl);
        NORET            lockid_t(const lockid_t& i);

        NORET            lockid_t(const user1_t& u);
        NORET            lockid_t(const user2_t& u);
        NORET            lockid_t(const user3_t& u);
        NORET            lockid_t(const user4_t& u);

        void            extract_extent(extid_t &e) const;
        void            extract_stid(stid_t &s) const;
        void            extract_lpid(lpid_t &p) const;
        void            extract_rid(rid_t &r) const;
        void            extract_kvl(kvl_t &k) const;
        void            extract_user1(user1_t &u) const;
        void            extract_user2(user2_t &u) const;
        void            extract_user3(user3_t &u) const;
        void            extract_user4(user4_t &u) const;

        bool            IsUserLock() const;

        void            truncate(name_space_t space);

        lockid_t&             operator=(const lockid_t& i);

    };

        ostream& operator<<(ostream& o, const lockid_t::user1_t& u);
        ostream& operator<<(ostream& o, const lockid_t::user2_t& u);
        ostream& operator<<(ostream& o, const lockid_t::user3_t& u);
        ostream& operator<<(ostream& o, const lockid_t::user4_t& u);

        istream& operator>>(istream& o, lockid_t::user1_t& u);
        istream& operator>>(istream& o, lockid_t::user2_t& u);
        istream& operator>>(istream& o, lockid_t::user3_t& u);
```

```
istream& operator>>(istream& o, lockid_t::user4_t& u);
```

DESCRIPTION

Locks are acquired implicitly by many **ss_m** methods. For those situations where more precise control of locking is desired, the following methods allow explicit locking and unlocking.

The class representing a generic lock is a lockid _t, described above. The SSM acquires locks on pages, extents, records, stores, and volumes. **The extent locks are NOT to be used by VASs,** simply because the extent-based structure of the SSM is likely to change in future releases.

lock(lvid, mode, duration, timeout)

lock(lockid, mode, duration, timeout)

The **lock** method is used to acquire a lock on volume, index, file or record. The first version of the method locks the volume specified by *lvid.* The second version locks the index, file or record specified by *lockid.* The *mode* parameter specifies the lock mode to acquire. Valid lock_mode_t values are listed in
`basics.h`. The *duration* parameter specifies how long the lock will be held. Valid values (among those listed in `basics.h`) are: t_instant, t_short and t_long. The *timeout* parameter specifies how long to wait for a lock.

unlock(lockid)

The **unlock** method releases the most recently acquired lock on the file, index, or record identified by *lockid.* Note, that only locks with duration **t_short** can be released before end-of-transaction.

query_lock(lockid, mode, implicit)

The **query_lock** method the mode of the lock held on *lockid* by the current transaction. The lock mode is returned in *mode* and will be **NL** (no lock) if not locked. If *implicit* is **false** then only explicit locks on *lockid* will be considered. For example, if file F is **SH** locked and a query is made about a record in F, the mode returned will be **NL. However, if** *implicit* is **true,** then **SH** would be returned for this example.

Lock Cache Control

Each transaction has a cache of recently acquired locks The following methods control the use of the cache. These are not supported methods and may be removed in later versions of the software. Note: that the methods only affect the transaction associated with the current thread.

set_lock_cache_enable(enable)

The **set_lock_cache_enable** method turns on the cache if *enable* is **true** and turns it off otherwise.

lock_cache_enabled(enabled)

The **lock_cache_enabled** method sets *enabled* to **true** if the lock cache is on.

Escalation

The lock manager will escalate from a record lock to a page lock, from a page lock to a store lock, and from a store lock to a volume lock, to reduce the number of locks in the table. You can control the thresholds for escalation throught the methods **get_escalation_thresholds** and **set_escalation_thresholds.** The default values are as follows:

record-to-page
5

page-to-store
25

store-to-volume
0

In all cases, a threshold of 0 prevents escalation.

When escalation is in use, it be prevented on selected volumes or other lock-able objects through the three **dont_escalate** methods. If the argument *passOnToDescendants is false,* locks acquired on objects below the volume (or given lockid) in the lock hierarchy will still be escalated according to the thresholds.

ERRORS

TODO

EXAMPLES

TODO

VERSION

This manual page applies to Version 2.0 of the Shore Storage Manager.

SPONSORSHIP

The Shore project is sponsored by the Advanced Research Project Agency, ARPA order number 018 (formerly 8230), monitored by the U.S. Army Research Laboratory under contract DAAB07-91-C-Q518. Further funding for this work was provided by DARPA through Rome Research Laboratory Contract No. F30602-97-2-0247.

COPYRIGHT

Copyright (c) 1994-1999, Computer Sciences Department, University of Wisconsin -- Madison. All Rights Reserved.

SEE ALSO

transaction(ssm), id(ssm), and **intro(ssm).**