```
**********************************************************
   50941 Fri Jun 19 17:15:05 2015
new/usr/src/uts/common/fs/zfs/dmu_objset.c
5981 Deadlock in dmu_objset_find_dp
**********************************************************
______unchanged_portion_omitted_

1743 static void
1744 dmu_objset_find_dp_cb(void *arg)
1745 {
1746         dmu_objset_find_ctx_t *dcp = arg;
1747         dsl_pool_t *dp = dcp->dc_dp;

1749         /*
1750          * We need to get a pool_config_lock here, as there are several
1751          * asssert(pool_config_held) down the stack. Getting a lock via
1752          * dsl_pool_config_enter is risky, as it might be stalled by a
1753          * pending writer. This would deadlock, as the write lock can
1754          * only be granted when our parent thread gives up the lock.
1755          * The _prio interface gives us priority over a pending writer.
1756          * On the other hand, we don't risk to stall any pending writers,
1757          * as the parent thread already holds a config lock. We give up
1758          * our lock before the parent does, so in effect we do not prolong
1759          * the waiting time for the writer.
1760          */
1761         dsl_pool_config_enter_prio(dp, FTAG);
1749         dsl_pool_config_enter(dp, FTAG);

1763         dmu_objset_find_dp_impl(dcp);

1765         dsl_pool_config_exit(dp, FTAG);
1766 }
______unchanged_portion_omitted_
```

****
   31464 Fri Jun 19 17:15:05 2015
new/usr/src/uts/common/fs/zfs/dsl_pool.c
5981 Deadlock in dmu_objset_find_dp
****
______*unchanged_portion_omitted_*

```
void
dsl_pool_config_enter_prio(dsl_pool_t *dp, void *tag)
{
        ASSERT(!rrw_held(&dp->dp_config_rwlock, RW_READER));
        rrw_enter_read_prio(&dp->dp_config_rwlock, tag);
}

void
#endif /* ! codereview */
dsl_pool_config_exit(dsl_pool_t *dp, void *tag)
{
        rrw_exit(&dp->dp_config_rwlock, tag);
}

boolean_t
dsl_pool_config_held(dsl_pool_t *dp)
{
        return (RRW_LOCK_HELD(&dp->dp_config_rwlock));
}

boolean_t
dsl_pool_config_held_writer(dsl_pool_t *dp)
{
        return (RRW_WRITE_HELD(&dp->dp_config_rwlock));
}
```

```
*********************************************************
   10974 Fri Jun 19 17:15:05 2015
new/usr/src/uts/common/fs/zfs/rrwlock.c
5981 Deadlock in dmu_objset_find_dp
*********************************************************
______unchanged_portion_omitted_

static void
rrw_enter_read_impl(rrwlock_t *rrl, boolean_t prio, void *tag)
void
rrw_enter_read(rrwlock_t *rrl, void *tag)
{
        mutex_enter(&rrl->rr_lock);
#if !defined(DEBUG) && defined(_KERNEL)
        if (rrl->rr_writer == NULL && !rrl->rr_writer_wanted &&
            !rrl->rr_track_all) {
                rrl->rr_anon_rcount.rc_count++;
                mutex_exit(&rrl->rr_lock);
                return;
        }
        DTRACE_PROBE(zfs__rrwfastpath__rdmiss);
#endif
        ASSERT(rrl->rr_writer != curthread);
        ASSERT(refcount_count(&rrl->rr_anon_rcount) >= 0);

        while (rrl->rr_writer != NULL || (rrl->rr_writer_wanted &&
            refcount_is_zero(&rrl->rr_anon_rcount) && !prio &&
            refcount_is_zero(&rrl->rr_anon_rcount) &&
            rrn_find(rrl) == NULL))
                cv_wait(&rrl->rr_cv, &rrl->rr_lock);

        if (rrl->rr_writer_wanted || rrl->rr_track_all) {
                /* may or may not be a re-entrant enter */
                rrn_add(rrl, tag);
                (void) refcount_add(&rrl->rr_linked_rcount, tag);
        } else {
                (void) refcount_add(&rrl->rr_anon_rcount, tag);
        }
        ASSERT(rrl->rr_writer == NULL);
        mutex_exit(&rrl->rr_lock);
}

void
rrw_enter_read(rrwlock_t *rrl, void *tag)
{
        rrw_enter_read_impl(rrl, B_FALSE, tag);
}

/*
 * take a read lock even if there are pending write lock requests. if we want
 * to take a lock reentrantly, but from different threads (that have a
 * relationship to each other), the normal detection mechanism to overrule
 * the pending writer does not work, so we have to give an explicit hint here.
 */
void
rrw_enter_read_prio(rrwlock_t *rrl, void *tag)
{
        rrw_enter_read_impl(rrl, B_TRUE, tag);
}


void
#endif /* ! codereview */
rrw_enter_write(rrwlock_t *rrl)
{
        mutex_enter(&rrl->rr_lock);
```

```
        ASSERT(rrl->rr_writer != curthread);

        while (refcount_count(&rrl->rr_anon_rcount) > 0 ||
            refcount_count(&rrl->rr_linked_rcount) > 0 ||
            rrl->rr_writer != NULL) {
                rrl->rr_writer_wanted = B_TRUE;
                cv_wait(&rrl->rr_cv, &rrl->rr_lock);
        }
        rrl->rr_writer_wanted = B_FALSE;
        rrl->rr_writer = curthread;
        mutex_exit(&rrl->rr_lock);
}

void
rrw_enter(rrwlock_t *rrl, krw_t rw, void *tag)
{
        if (rw == RW_READER)
                rrw_enter_read(rrl, tag);
        else
                rrw_enter_write(rrl);
}

void
rrw_exit(rrwlock_t *rrl, void *tag)
{
        mutex_enter(&rrl->rr_lock);
#if !defined(DEBUG) && defined(_KERNEL)
        if (!rrl->rr_writer && rrl->rr_linked_rcount.rc_count == 0) {
                rrl->rr_anon_rcount.rc_count--;
                if (rrl->rr_anon_rcount.rc_count == 0)
                        cv_broadcast(&rrl->rr_cv);
                mutex_exit(&rrl->rr_lock);
                return;
        }
        DTRACE_PROBE(zfs__rrwfastpath__exitmiss);
#endif
        ASSERT(!refcount_is_zero(&rrl->rr_anon_rcount) ||
            !refcount_is_zero(&rrl->rr_linked_rcount) ||
            rrl->rr_writer != NULL);

        if (rrl->rr_writer == NULL) {
                int64_t count;
                if (rrn_find_and_remove(rrl, tag)) {
                        count = refcount_remove(&rrl->rr_linked_rcount, tag);
                } else {
                        ASSERT(!rrl->rr_track_all);
                        count = refcount_remove(&rrl->rr_anon_rcount, tag);
                }
                if (count == 0)
                        cv_broadcast(&rrl->rr_cv);
        } else {
                ASSERT(rrl->rr_writer == curthread);
                ASSERT(refcount_is_zero(&rrl->rr_anon_rcount) &&
                    refcount_is_zero(&rrl->rr_linked_rcount));
                rrl->rr_writer = NULL;
                cv_broadcast(&rrl->rr_cv);
        }
        mutex_exit(&rrl->rr_lock);
}

/*
 * If the lock was created with track_all, rrw_held(RW_READER) will return
 * B_TRUE iff the current thread has the lock for reader.  Otherwise it may
 * return B_TRUE if any thread has the lock for reader.
 */
boolean_t
```

```
rrw_held(rrwlock_t *rrl, krw_t rw)
{
        boolean_t held;

        mutex_enter(&rrl->rr_lock);
        if (rw == RW_WRITER) {
                held = (rrl->rr_writer == curthread);
        } else {
                held = (!refcount_is_zero(&rrl->rr_anon_rcount) ||
                    rrn_find(rrl) != NULL);
        }
        mutex_exit(&rrl->rr_lock);

        return (held);
}

void
rrw_tsd_destroy(void *arg)
{
        rrw_node_t *rn = arg;
        if (rn != NULL) {
                panic("thread %p terminating with rrw lock %p held",
                    (void *)curthread, (void *)rn->rn_rrl);
        }
}

/*
 * A reader-mostly lock implementation, tuning above reader-writer locks
 * for hightly parallel read acquisitions, while pessimizing writes.
 *
 * The idea is to split single busy lock into array of locks, so that
 * each reader can lock only one of them for read, depending on result
 * of simple hash function.  That proportionally reduces lock congestion.
 * Writer same time has to sequentially aquire write on all the locks.
 * That makes write aquisition proportionally slower, but in places where
 * it is used (filesystem unmount) performance is not critical.
 *
 * All the functions below are direct wrappers around functions above.
 */
void
rrm_init(rrmlock_t *rrl, boolean_t track_all)
{
        int i;

        for (i = 0; i < RRM_NUM_LOCKS; i++)
                rrw_init(&rrl->locks[i], track_all);
}

void
rrm_destroy(rrmlock_t *rrl)
{
        int i;

        for (i = 0; i < RRM_NUM_LOCKS; i++)
                rrw_destroy(&rrl->locks[i]);
}

void
rrm_enter(rrmlock_t *rrl, krw_t rw, void *tag)
{
        if (rw == RW_READER)
                rrm_enter_read(rrl, tag);
        else
                rrm_enter_write(rrl);
}
```

```
/*
 * This maps the current thread to a specific lock.  Note that the lock
 * must be released by the same thread that acquired it.  We do this
 * mapping by taking the thread pointer mod a prime number.  We examine
 * only the low 32 bits of the thread pointer, because 32-bit division
 * is faster than 64-bit division, and the high 32 bits have little
 * entropy anyway.
 */
#define RRM_TD_LOCK()   (((uint32_t)(uintptr_t)(curthread)) % RRM_NUM_LOCKS)

void
rrm_enter_read(rrmlock_t *rrl, void *tag)
{
        rrw_enter_read(&rrl->locks[RRM_TD_LOCK()], tag);
}

void
rrm_enter_write(rrmlock_t *rrl)
{
        int i;

        for (i = 0; i < RRM_NUM_LOCKS; i++)
                rrw_enter_write(&rrl->locks[i]);
}

void
rrm_exit(rrmlock_t *rrl, void *tag)
{
        int i;

        if (rrl->locks[0].rr_writer == curthread) {
                for (i = 0; i < RRM_NUM_LOCKS; i++)
                        rrw_exit(&rrl->locks[i], tag);
        } else {
                rrw_exit(&rrl->locks[RRM_TD_LOCK()], tag);
        }
}

boolean_t
rrm_held(rrmlock_t *rrl, krw_t rw)
{
        if (rw == RW_WRITER) {
                return (rrw_held(&rrl->locks[0], rw));
        } else {
                return (rrw_held(&rrl->locks[RRM_TD_LOCK()], rw));
        }
}
```

```
**********************************************************
    5542 Fri Jun 19 17:15:05 2015
new/usr/src/uts/common/fs/zfs/sys/dsl_pool.h
5981 Deadlock in dmu_objset_find_dp
**********************************************************
______unchanged_portion_omitted_
```

```
int dsl_pool_init(spa_t *spa, uint64_t txg, dsl_pool_t **dpp);
int dsl_pool_open(dsl_pool_t *dp);
void dsl_pool_close(dsl_pool_t *dp);
dsl_pool_t *dsl_pool_create(spa_t *spa, nvlist_t *zplprops, uint64_t txg);
void dsl_pool_sync(dsl_pool_t *dp, uint64_t txg);
void dsl_pool_sync_done(dsl_pool_t *dp, uint64_t txg);
int dsl_pool_sync_context(dsl_pool_t *dp);
uint64_t dsl_pool_adjustedsize(dsl_pool_t *dp, boolean_t netfree);
uint64_t dsl_pool_adjustedfree(dsl_pool_t *dp, boolean_t netfree);
void dsl_pool_dirty_space(dsl_pool_t *dp, int64_t space, dmu_tx_t *tx);
void dsl_pool_undirty_space(dsl_pool_t *dp, int64_t space, uint64_t txg);
void dsl_free(dsl_pool_t *dp, uint64_t txg, const blkptr_t *bpp);
void dsl_free_sync(zio_t *pio, dsl_pool_t *dp, uint64_t txg,
    const blkptr_t *bpp);
void dsl_pool_create_origin(dsl_pool_t *dp, dmu_tx_t *tx);
void dsl_pool_upgrade_clones(dsl_pool_t *dp, dmu_tx_t *tx);
void dsl_pool_upgrade_dir_clones(dsl_pool_t *dp, dmu_tx_t *tx);
void dsl_pool_mos_diduse_space(dsl_pool_t *dp,
    int64_t used, int64_t comp, int64_t uncomp);
void dsl_pool_config_enter(dsl_pool_t *dp, void *tag);
void dsl_pool_config_enter_prio(dsl_pool_t *dp, void *tag);
#endif /* ! codereview */
void dsl_pool_config_exit(dsl_pool_t *dp, void *tag);
boolean_t dsl_pool_config_held(dsl_pool_t *dp);
boolean_t dsl_pool_config_held_writer(dsl_pool_t *dp);
boolean_t dsl_pool_need_dirty_delay(dsl_pool_t *dp);

taskq_t *dsl_pool_vnrele_taskq(dsl_pool_t *dp);

int dsl_pool_user_hold(dsl_pool_t *dp, uint64_t dsobj,
    const char *tag, uint64_t now, dmu_tx_t *tx);
int dsl_pool_user_release(dsl_pool_t *dp, uint64_t dsobj,
    const char *tag, dmu_tx_t *tx);
void dsl_pool_clean_tmp_userrefs(dsl_pool_t *dp);
int dsl_pool_open_special_dir(dsl_pool_t *dp, const char *name, dsl_dir_t **);
int dsl_pool_hold(const char *name, void *tag, dsl_pool_t **dp);
void dsl_pool_rele(dsl_pool_t *dp, void *tag);

#ifdef  __cplusplus
}
#endif

#endif /* _SYS_DSL_POOL_H */
```

```
**********************************************************
    3572 Fri Jun 19 17:15:05 2015
new/usr/src/uts/common/fs/zfs/sys/rrwlock.h
5981 Deadlock in dmu_objset_find_dp
**********************************************************
______unchanged_portion_omitted_

/*
 * 'tag' is used in reference counting tracking.  The
 * 'tag' must be the same in a rrw_enter() as in its
 * corresponding rrw_exit().
 */
void rrw_init(rrwlock_t *rrl, boolean_t track_all);
void rrw_destroy(rrwlock_t *rrl);
void rrw_enter(rrwlock_t *rrl, krw_t rw, void *tag);
void rrw_enter_read(rrwlock_t *rrl, void *tag);
void rrw_enter_read_prio(rrwlock_t *rrl, void *tag);
#endif /* ! codereview */
void rrw_enter_write(rrwlock_t *rrl);
void rrw_exit(rrwlock_t *rrl, void *tag);
boolean_t rrw_held(rrwlock_t *rrl, krw_t rw);
void rrw_tsd_destroy(void *arg);

#define RRW_READ_HELD(x)        rrw_held(x, RW_READER)
#define RRW_WRITE_HELD(x)       rrw_held(x, RW_WRITER)
#define RRW_LOCK_HELD(x) \
        (rrw_held(x, RW_WRITER) || rrw_held(x, RW_READER))

/*
 * A reader-mostly lock implementation, tuning above reader-writer locks
 * for hightly parallel read acquisitions, pessimizing write acquisitions.
 *
 * This should be a prime number.  See comment in rrwlock.c near
 * RRM_TD_LOCK() for details.
 */
#define RRM_NUM_LOCKS           17
typedef struct rrmlock {
        rrwlock_t       locks[RRM_NUM_LOCKS];
} rrmlock_t;

void rrm_init(rrmlock_t *rrl, boolean_t track_all);
void rrm_destroy(rrmlock_t *rrl);
void rrm_enter(rrmlock_t *rrl, krw_t rw, void *tag);
void rrm_enter_read(rrmlock_t *rrl, void *tag);
void rrm_enter_write(rrmlock_t *rrl);
void rrm_exit(rrmlock_t *rrl, void *tag);
boolean_t rrm_held(rrmlock_t *rrl, krw_t rw);

#define RRM_READ_HELD(x)        rrm_held(x, RW_READER)
#define RRM_WRITE_HELD(x)       rrm_held(x, RW_WRITER)
#define RRM_LOCK_HELD(x) \
        (rrm_held(x, RW_WRITER) || rrm_held(x, RW_READER))

#ifdef  __cplusplus
}
#endif

#endif  /* _SYS_RR_RW_LOCK_H */
```