

new/usr/src/uts/common/fs/zfs/dmu\_objset.c

1

\*\*\*\*\*

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         \* assassert(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\_\_\_\_\_

new/usr/src/uts/common/fs/zfs/dsl\_pool.c

1

```
*****  
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_____
```

```
1048 void  
1049 dsl_pool_config_enter_prio(dsl_pool_t *dp, void *tag)  
1050 {  
1051     ASSERT(!rrw_held(&dp->dp_config_rwlock, RW_READER));  
1052     rrw_enter_read_prio(&dp->dp_config_rwlock, tag);  
1053 }  
  
1055 void  
1056 #endif /* !codereview */  
1057 dsl_pool_config_exit(dsl_pool_t *dp, void *tag)  
1058 {  
1059     rrw_exit(&dp->dp_config_rwlock, tag);  
1060 }  
  
1062 boolean_t  
1063 dsl_pool_config_held(dsl_pool_t *dp)  
1064 {  
1065     return (RRW_LOCK_HELD(&dp->dp_config_rwlock));  
1066 }  
  
1068 boolean_t  
1069 dsl_pool_config_held_writer(dsl_pool_t *dp)  
1070 {  
1071     return (RRW_WRITE_HELD(&dp->dp_config_rwlock));  
1072 }
```

```

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

162 static void
163 rrw_enter_read_impl(rrwlock_t *rrl, boolean_t prio, void *tag)
164 {
165     mutex_enter(&rrl->rr_lock);
166 #if !defined(DEBUG) && defined(_KERNEL)
167     if (rrl->rr_writer == NULL && !rrl->rr_writer_wanted &&
168         !rrl->rr_track_all) {
169         rrl->rr_anon_rcount.rc_count++;
170         mutex_exit(&rrl->rr_lock);
171         return;
172     }
173     DTRACE_PROBE(zfs_rrwfastpath_rdmis);
174 #endif
175     ASSERT(rrl->rr_writer != curthread);
176     ASSERT(refcount_count(&rrl->rr_anon_rcount) >= 0);

178     while (rrl->rr_writer != NULL || (rrl->rr_writer_wanted &&
179         refcount_is_zero(&rrl->rr_anon_rcount) && !prio &&
180         refcount_is_zero(&rrl->rr_linked_rcount) &&
181         rrl->rr_track_all == NULL)) {
182         cv_wait(&rrl->rr_cv, &rrl->rr_lock);
183     }
184     if (rrl->rr_writer_wanted || rrl->rr_track_all) {
185         /* may or may not be a re-entrant enter */
186         rrl->rr_track_all++;
187         (void) refcount_add(&rrl->rr_linked_rcount, tag);
188     } else {
189         (void) refcount_add(&rrl->rr_anon_rcount, tag);
190     }
191     ASSERT(rrl->rr_writer == NULL);
192     mutex_exit(&rrl->rr_lock);

194 void
195 rrw_enter_read(rrwlock_t *rrl, void *tag)
196 {
197     rrw_enter_read_impl(rrl, B_FALSE, tag);
198 }

200 /*
201  * take a read lock even if there are pending write lock requests. if we want
202  * to take a lock reentrantly, but from different threads (that have a
203  * relationship to each other), the normal detection mechanism to overrule
204  * the pending writer does not work, so we have to give an explicit hint here.
205  */
206 void
207 rrw_enter_read_prio(rrwlock_t *rrl, void *tag)
208 {
209     rrw_enter_read_impl(rrl, B_TRUE, tag);
210 }

213 void
214 #endif /* !codereview */
215 rrw_enter_write(rrwlock_t *rrl)
216 {
217     mutex_enter(&rrl->rr_lock);

```

```

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

220     while (refcount_count(&rrl->rr_anon_rcount) > 0 ||
221         refcount_count(&rrl->rr_linked_rcount) > 0 ||
222         rrl->rr_writer != NULL) {
223         rrl->rr_writer_wanted = B_TRUE;
224         cv_wait(&rrl->rr_cv, &rrl->rr_lock);
225     }
226     rrl->rr_writer_wanted = B_FALSE;
227     rrl->rr_writer = curthread;
228     mutex_exit(&rrl->rr_lock);
229 }

231 void
232 rrw_enter(rrwlock_t *rrl, krw_t rw, void *tag)
233 {
234     if (rw == RW_READER)
235         rrw_enter_read(rrl, tag);
236     else
237         rrw_enter_write(rrl);
238 }

240 void
241 rrw_exit(rrwlock_t *rrl, void *tag)
242 {
243     mutex_enter(&rrl->rr_lock);
244 #if !defined(DEBUG) && defined(_KERNEL)
245     if (!rrl->rr_writer && rrl->rr_linked_rcount.rc_count == 0) {
246         rrl->rr_anon_rcount.rc_count--;
247         if (rrl->rr_anon_rcount.rc_count == 0)
248             cv_broadcast(&rrl->rr_cv);
249         mutex_exit(&rrl->rr_lock);
250         return;
251     }
252     DTRACE_PROBE(zfs_rrwfastpath_exitmiss);
253 #endif
254     ASSERT(!refcount_is_zero(&rrl->rr_anon_rcount) ||
255         !refcount_is_zero(&rrl->rr_linked_rcount) ||
256         rrl->rr_writer != NULL);

258     if (rrl->rr_writer == NULL) {
259         int64_t count;
260         if (rrl->rr_track_all && rrl->rr_track_all == 0) {
261             count = refcount_remove(&rrl->rr_linked_rcount, tag);
262         } else {
263             ASSERT(!rrl->rr_track_all);
264             count = refcount_remove(&rrl->rr_anon_rcount, tag);
265         }
266         if (count == 0)
267             cv_broadcast(&rrl->rr_cv);
268     } else {
269         ASSERT(rrl->rr_writer == curthread);
270         ASSERT(refcount_is_zero(&rrl->rr_anon_rcount) &&
271             refcount_is_zero(&rrl->rr_linked_rcount));
272         rrl->rr_writer = NULL;
273         cv_broadcast(&rrl->rr_cv);
274     }
275     mutex_exit(&rrl->rr_lock);
276 }

278 /*
279  * If the lock was created with track_all, rrw_held(RW_READER) will return
280  * B_TRUE iff the current thread has the lock for reader. Otherwise it may
281  * return B_TRUE if any thread has the lock for reader.
282  */
283 boolean_t

```

```

284 rrw_held(rrwlock_t *rrl, krw_t rw)
285 {
286     boolean_t held;

288     mutex_enter(&rrl->rr_lock);
289     if (rw == RW_WRITER) {
290         held = (rrl->rr_writer == curthread);
291     } else {
292         held = (!refcount_is_zero(&rrl->rr_anon_rcount) ||
293             rrn_find(rrl) != NULL);
294     }
295     mutex_exit(&rrl->rr_lock);

297     return (held);
298 }

300 void
301 rrw_tsd_destroy(void *arg)
302 {
303     rrw_node_t *rn = arg;
304     if (rn != NULL) {
305         panic("thread %p terminating with rrw lock %p held",
306             (void *)curthread, (void *)rn->rn_rrl);
307     }
308 }

310 /*
311  * A reader-mostly lock implementation, tuning above reader-writer locks
312  * for highly parallel read acquisitions, while pessimizing writes.
313  *
314  * The idea is to split single busy lock into array of locks, so that
315  * each reader can lock only one of them for read, depending on result
316  * of simple hash function. That proportionally reduces lock congestion.
317  * Writer same time has to sequentially acquire write on all the locks.
318  * That makes write acquisition proportionally slower, but in places where
319  * it is used (filesystem unmount) performance is not critical.
320  *
321  * All the functions below are direct wrappers around functions above.
322  */
323 void
324 rrm_init(rrmlock_t *rrl, boolean_t track_all)
325 {
326     int i;

328     for (i = 0; i < RRM_NUM_LOCKS; i++)
329         rrw_init(&rrl->locks[i], track_all);
330 }

332 void
333 rrm_destroy(rrmlock_t *rrl)
334 {
335     int i;

337     for (i = 0; i < RRM_NUM_LOCKS; i++)
338         rrw_destroy(&rrl->locks[i]);
339 }

341 void
342 rrm_enter(rrmlock_t *rrl, krw_t rw, void *tag)
343 {
344     if (rw == RW_READER)
345         rrm_enter_read(rrl, tag);
346     else
347         rrm_enter_write(rrl);
348 }

```

```

350 /*
351  * This maps the current thread to a specific lock. Note that the lock
352  * must be released by the same thread that acquired it. We do this
353  * mapping by taking the thread pointer mod a prime number. We examine
354  * only the low 32 bits of the thread pointer, because 32-bit division
355  * is faster than 64-bit division, and the high 32 bits have little
356  * entropy anyway.
357  */
358 #define RRM_TD_LOCK() (((uint32_t)(uintptr_t)(curthread)) % RRM_NUM_LOCKS)

360 void
361 rrm_enter_read(rrmlock_t *rrl, void *tag)
362 {
363     rrw_enter_read(&rrl->locks[RRM_TD_LOCK()], tag);
364 }

366 void
367 rrm_enter_write(rrmlock_t *rrl)
368 {
369     int i;

371     for (i = 0; i < RRM_NUM_LOCKS; i++)
372         rrw_enter_write(&rrl->locks[i]);
373 }

375 void
376 rrm_exit(rrmlock_t *rrl, void *tag)
377 {
378     int i;

380     if (rrl->locks[0].rr_writer == curthread) {
381         for (i = 0; i < RRM_NUM_LOCKS; i++)
382             rrw_exit(&rrl->locks[i], tag);
383     } else {
384         rrw_exit(&rrl->locks[RRM_TD_LOCK()], tag);
385     }
386 }

388 boolean_t
389 rrm_held(rrmlock_t *rrl, krw_t rw)
390 {
391     if (rw == RW_WRITER) {
392         return (rrw_held(&rrl->locks[0], rw));
393     } else {
394         return (rrw_held(&rrl->locks[RRM_TD_LOCK()], rw));
395     }
396 }

```

```
*****
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_____

135 int dsl_pool_init(spa_t *spa, uint64_t txg, dsl_pool_t **dpp);
136 int dsl_pool_open(dsl_pool_t *dp);
137 void dsl_pool_close(dsl_pool_t *dp);
138 dsl_pool_t *dsl_pool_create(spa_t *spa, nvlist_t *zplprops, uint64_t txg);
139 void dsl_pool_sync(dsl_pool_t *dp, uint64_t txg);
140 void dsl_pool_sync_done(dsl_pool_t *dp, uint64_t txg);
141 int dsl_pool_sync_context(dsl_pool_t *dp);
142 uint64_t dsl_pool_adjustedsize(dsl_pool_t *dp, boolean_t netfree);
143 uint64_t dsl_pool_adjustedfree(dsl_pool_t *dp, boolean_t netfree);
144 void dsl_pool_dirty_space(dsl_pool_t *dp, int64_t space, dmu_tx_t *tx);
145 void dsl_pool_undirty_space(dsl_pool_t *dp, int64_t space, uint64_t txg);
146 void dsl_free(dsl_pool_t *dp, uint64_t txg, const blkptr_t *bpp);
147 void dsl_free_sync(zio_t *pio, dsl_pool_t *dp, uint64_t txg,
148     const blkptr_t *bpp);
149 void dsl_pool_create_origin(dsl_pool_t *dp, dmu_tx_t *tx);
150 void dsl_pool_upgrade_clones(dsl_pool_t *dp, dmu_tx_t *tx);
151 void dsl_pool_upgrade_dir_clones(dsl_pool_t *dp, dmu_tx_t *tx);
152 void dsl_pool_mos_diduse_space(dsl_pool_t *dp,
153     int64_t used, int64_t comp, int64_t uncomp);
154 void dsl_pool_config_enter(dsl_pool_t *dp, void *tag);
155 void dsl_pool_config_enter_prio(dsl_pool_t *dp, void *tag);
156 #endif /* !codereview */
157 void dsl_pool_config_exit(dsl_pool_t *dp, void *tag);
158 boolean_t dsl_pool_config_held(dsl_pool_t *dp);
159 boolean_t dsl_pool_config_held_writer(dsl_pool_t *dp);
160 boolean_t dsl_pool_need_dirty_delay(dsl_pool_t *dp);

162 taskq_t *dsl_pool_vnrele_taskq(dsl_pool_t *dp);

164 int dsl_pool_user_hold(dsl_pool_t *dp, uint64_t dsobj,
165     const char *tag, uint64_t now, dmu_tx_t *tx);
166 int dsl_pool_user_release(dsl_pool_t *dp, uint64_t dsobj,
167     const char *tag, dmu_tx_t *tx);
168 void dsl_pool_clean_tmp_userrefs(dsl_pool_t *dp);
169 int dsl_pool_open_special_dir(dsl_pool_t *dp, const char *name, dsl_dir_t **);
170 int dsl_pool_hold(const char *name, void *tag, dsl_pool_t **dp);
171 void dsl_pool_rele(dsl_pool_t *dp, void *tag);

173 #ifdef __cplusplus
174 }
175 #endif

177 #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_____
```

```
64 /*
65  * 'tag' is used in reference counting tracking. The
66  * 'tag' must be the same in a rrw_enter() as in its
67  * corresponding rrw_exit().
68  */
69 void rrw_init(rrwlock_t *rri, boolean_t track_all);
70 void rrw_destroy(rrwlock_t *rri);
71 void rrw_enter(rrwlock_t *rri, krw_t rw, void *tag);
72 void rrw_enter_read(rrwlock_t *rri, void *tag);
73 void rrw_enter_read_prio(rrwlock_t *rri, void *tag);
74 #endif /* !codereview */
75 void rrw_enter_write(rrwlock_t *rri);
76 void rrw_exit(rrwlock_t *rri, void *tag);
77 boolean_t rrw_held(rrwlock_t *rri, krw_t rw);
78 void rrw_tsd_destroy(void *arg);

80 #define RRW_READ_HELD(x)      rrw_held(x, RW_READER)
81 #define RRW_WRITE_HELD(x)    rrw_held(x, RW_WRITER)
82 #define RRW_LOCK_HELD(x) \
83     (rrw_held(x, RW_WRITER) || rrw_held(x, RW_READER))

85 /*
86  * A reader-mostly lock implementation, tuning above reader-writer locks
87  * for highly parallel read acquisitions, pessimizing write acquisitions.
88  *
89  * This should be a prime number. See comment in rrwlock.c near
90  * RRM_TD_LOCK() for details.
91  */
92 #define RRM_NUM_LOCKS      17
93 typedef struct rrmlock {
94     rrwlock_t      locks[RRM_NUM_LOCKS];
95 } rrmlock_t;

97 void rrm_init(rrmlock_t *rri, boolean_t track_all);
98 void rrm_destroy(rrmlock_t *rri);
99 void rrm_enter(rrmlock_t *rri, krw_t rw, void *tag);
100 void rrm_enter_read(rrmlock_t *rri, void *tag);
101 void rrm_enter_write(rrmlock_t *rri);
102 void rrm_exit(rrmlock_t *rri, void *tag);
103 boolean_t rrm_held(rrmlock_t *rri, krw_t rw);

105 #define RRM_READ_HELD(x)      rrm_held(x, RW_READER)
106 #define RRM_WRITE_HELD(x)    rrm_held(x, RW_WRITER)
107 #define RRM_LOCK_HELD(x) \
108     (rrm_held(x, RW_WRITER) || rrm_held(x, RW_READER))

110 #ifdef __cplusplus
111 }
112 #endif

114 #endif /* _SYS_RR_RW_LOCK_H */
```