****
49065 Tue Feb 14 19:22:19 2012
new/usr/src/uts/common/fs/zfs/zvol.c
2130 zvol DKIOCFREE uses nested DMU transactions
****
_____**unchanged_portion_omitted_**

```
1558 /*
1559  * Dirtbag ioctls to support mkfs(1M) for UFS filesystems.  See dkio(7I).
1560  * Also a dirtbag dkio ioctl for unmap/free-block functionality.
1561  */
1562 /*ARGSUSED*/
1563 int
1564 zvol_ioctl(dev_t dev, int cmd, intptr_t arg, int flag, cred_t *cr, int *rvalp)
1565 {
1566         zvol_state_t *zv;
1567         struct dk_cinfo dki;
1568         struct dk_minfo dkm;
1569         struct dk_callback *dkc;
1570         int error = 0;
1571         rl_t *rl;

1573         mutex_enter(&zfsdev_state_lock);

1575         zv = zfsdev_get_soft_state(getminor(dev), ZSST_ZVOL);

1577         if (zv == NULL) {
1578                 mutex_exit(&zfsdev_state_lock);
1579                 return (ENXIO);
1580         }
1581         ASSERT(zv->zv_total_opens > 0);

1583         switch (cmd) {

1585         case DKIOCINFO:
1586                 bzero(&dki, sizeof (dki));
1587                 (void) strcpy(dki.dki_cname, "zvol");
1588                 (void) strcpy(dki.dki_dname, "zvol");
1589                 dki.dki_ctype = DKC_UNKNOWN;
1590                 dki.dki_unit = getminor(dev);
1591                 dki.dki_maxtransfer = 1 << (SPA_MAXBLOCKSHIFT - zv->zv_min_bs);
1592                 mutex_exit(&zfsdev_state_lock);
1593                 if (ddi_copyout(&dki, (void *)arg, sizeof (dki), flag))
1594                         error = EFAULT;
1595                 return (error);

1597         case DKIOCGMEDIAINFO:
1598                 bzero(&dkm, sizeof (dkm));
1599                 dkm.dki_lbsize = 1U << zv->zv_min_bs;
1600                 dkm.dki_capacity = zv->zv_volsize >> zv->zv_min_bs;
1601                 dkm.dki_media_type = DK_UNKNOWN;
1602                 mutex_exit(&zfsdev_state_lock);
1603                 if (ddi_copyout(&dkm, (void *)arg, sizeof (dkm), flag))
1604                         error = EFAULT;
1605                 return (error);

1607         case DKIOCGETEFI:
1608                 {
1609                         uint64_t vs = zv->zv_volsize;
1610                         uint8_t bs = zv->zv_min_bs;

1612                         mutex_exit(&zfsdev_state_lock);
1613                         error = zvol_getefi((void *)arg, flag, vs, bs);
1614                         return (error);
1615                 }
```



```
1617         case DKIOCFLUSHWRITECACHE:
1618                 dkc = (struct dk_callback *)arg;
1619                 mutex_exit(&zfsdev_state_lock);
1620                 zil_commit(zv->zv_zilog, ZVOL_OBJ);
1621                 if ((flag & FKIOCTL) && dkc != NULL && dkc->dkc_callback) {
1622                         (*dkc->dkc_callback)(dkc->dkc_cookie, error);
1623                         error = 0;
1624                 }
1625                 return (error);

1627         case DKIOCGETWCE:
1628                 {
1629                         int wce = (zv->zv_flags & ZVOL_WCE) ? 1 : 0;
1630                         if (ddi_copyout(&wce, (void *)arg, sizeof (int),
1631                             flag))
1632                                 error = EFAULT;
1633                         break;
1634                 }
1635         case DKIOCSETWCE:
1636                 {
1637                         int wce;
1638                         if (ddi_copyin((void *)arg, &wce, sizeof (int),
1639                             flag)) {
1640                                 error = EFAULT;
1641                                 break;
1642                         }
1643                         if (wce) {
1644                                 zv->zv_flags |= ZVOL_WCE;
1645                                 mutex_exit(&zfsdev_state_lock);
1646                         } else {
1647                                 zv->zv_flags &= ~ZVOL_WCE;
1648                                 mutex_exit(&zfsdev_state_lock);
1649                                 zil_commit(zv->zv_zilog, ZVOL_OBJ);
1650                         }
1651                         return (0);
1652                 }

1654         case DKIOCGGEOM:
1655         case DKIOCGVTOC:
1656                 /*
1657                  * commands using these (like prtvtoc) expect ENOTSUP
1658                  * since we're emulating an EFI label
1659                  */
1660                 error = ENOTSUP;
1661                 break;

1663         case DKIOCDUMPINIT:
1664                 rl = zfs_range_lock(&zv->zv_znode, 0, zv->zv_volsize,
1665                     RL_WRITER);
1666                 error = zvol_dumpify(zv);
1667                 zfs_range_unlock(rl);
1668                 break;

1670         case DKIOCDUMPFINI:
1671                 if (!(zv->zv_flags & ZVOL_DUMPIFIED))
1672                         break;
1673                 rl = zfs_range_lock(&zv->zv_znode, 0, zv->zv_volsize,
1674                     RL_WRITER);
1675                 error = zvol_dump_fini(zv);
1676                 zfs_range_unlock(rl);
1677                 break;

1679         case DKIOCFREE:
1680         {
1681                 dkioc_free_t df;
1682                 dmu_tx_t *tx;
```

```
1684                if (ddi_copyin((void *)arg, &df, sizeof (df), flag)) {
1685                        error = EFAULT;
1686                        break;
1687                }

1689                /*
1690                 * Apply Postel's Law to length-checking.  If they overshoot,
1691                 * just blank out until the end, if there's a need to blank
1692                 * out anything.
1693                 */
1694                if (df.df_start >= zv->zv_volsize)
1695                        break;  /* No need to do anything... */
1696                if (df.df_start + df.df_length > zv->zv_volsize)
1697                        df.df_length = DMU_OBJECT_END;

1699                rl = zfs_range_lock(&zv->zv_znode, df.df_start, df.df_length,
1700                    RL_WRITER);
1701                tx = dmu_tx_create(zv->zv_objset);
1702                error = dmu_tx_assign(tx, TXG_WAIT);
1703                if (error != 0) {
1704                        dmu_tx_abort(tx);
1705                } else {
1706                        zvol_log_truncate(zv, tx, df.df_start,
1707                            df.df_length, B_TRUE);
1708                        dmu_tx_commit(tx);
1709                        error = dmu_free_long_range(zv->zv_objset, ZVOL_OBJ,
1710                            df.df_start, df.df_length);
1710                        dmu_tx_commit(tx);
1711                }

1713                zfs_range_unlock(rl);

1715                if (error == 0) {
1716                        /*
1717                         * If the write-cache is disabled or 'sync' property
1718                         * is set to 'always' then treat this as a synchronous
1719                         * operation (i.e. commit to zil).
1720                         */
1721                        if (!(zv->zv_flags & ZVOL_WCE) ||
1722                            (zv->zv_objset->os_sync == ZFS_SYNC_ALWAYS))
1723                                zil_commit(zv->zv_zilog, ZVOL_OBJ);

1725                        /*
1726                         * If the caller really wants synchronous writes, and
1727                         * can't wait for them, don't return until the write
1728                         * is done.
1729                         */
1730                        if (df.df_flags & DF_WAIT_SYNC) {
1731                                txg_wait_synced(
1732                                    dmu_objset_pool(zv->zv_objset), 0);
1733                        }
1734                }
1735                break;
1736        }

1738        default:
1739                error = ENOTTY;
1740                break;

1742        }
1743        mutex_exit(&zfsdev_state_lock);
1744        return (error);
1745 }
```

_____***unchanged_portion_omitted_***