```
**********************************************************
   53159 Tue Nov 26 16:39:06 2013
new/usr/src/uts/common/fs/zfs/zvol.c
3580 Want zvols to return volblocksize when queried for physical block size
Reviewed by: Matthew Ahrens <mahrens@delphix.com>
Reviewed by: Dan Kimmel <dan.kimmel@delphix.com>
Reviewed by: Adam Leventhal <ahl@delphix.com>
Reviewed by: Christopher Siden <christopher.siden@delphix.com>
**********************************************************
_____unchanged_portion_omitted_

/*
 * Dirtbag ioctls to support mkfs(1M) for UFS filesystems.  See dkio(7I).
 * Also a dirtbag dkio ioctl for unmap/free-block functionality.
 */
/*ARGSUSED*/
int
zvol_ioctl(dev_t dev, int cmd, intptr_t arg, int flag, cred_t *cr, int *rvalp)
{
        zvol_state_t *zv;
        struct dk_cinfo dki;
        struct dk_minfo dkm;
        struct dk_callback *dkc;
        int error = 0;
        rl_t *rl;

        mutex_enter(&zfsdev_state_lock);

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

        if (zv == NULL) {
                mutex_exit(&zfsdev_state_lock);
                return (SET_ERROR(ENXIO));
        }
        ASSERT(zv->zv_total_opens > 0);

        switch (cmd) {

        case DKIOCINFO:
        {
                struct dk_cinfo dki;

                bzero(&dki, sizeof (dki));
                (void) strcpy(dki.dki_cname, "zvol");
                (void) strcpy(dki.dki_dname, "zvol");
                dki.dki_ctype = DKC_UNKNOWN;
                dki.dki_unit = getminor(dev);
                dki.dki_maxtransfer = 1 << (SPA_MAXBLOCKSHIFT - zv->zv_min_bs);
                mutex_exit(&zfsdev_state_lock);
                if (ddi_copyout(&dki, (void *)arg, sizeof (dki), flag))
                        error = SET_ERROR(EFAULT);
                return (error);
        }

        case DKIOCGMEDIAINFO:
        {
                struct dk_minfo dkm;

                bzero(&dkm, sizeof (dkm));
                dkm.dki_lbsize = 1U << zv->zv_min_bs;
                dkm.dki_capacity = zv->zv_volsize >> zv->zv_min_bs;
                dkm.dki_media_type = DK_UNKNOWN;
                mutex_exit(&zfsdev_state_lock);
                if (ddi_copyout(&dkm, (void *)arg, sizeof (dkm), flag))
                        error = SET_ERROR(EFAULT);
                return (error);
        }

        case DKIOCGMEDIAINFOEXT:
        {
                struct dk_minfo_ext dkmext;

                bzero(&dkmext, sizeof (dkmext));
                dkmext.dki_lbsize = 1U << zv->zv_min_bs;
                dkmext.dki_pbsize = zv->zv_volblocksize;
                dkmext.dki_capacity = zv->zv_volsize >> zv->zv_min_bs;
                dkmext.dki_media_type = DK_UNKNOWN;
                mutex_exit(&zfsdev_state_lock);
                if (ddi_copyout(&dkmext, (void *)arg, sizeof (dkmext), flag))
                        error = SET_ERROR(EFAULT);
                return (error);
        }

        case DKIOCGETEFI:
        {
                uint64_t vs = zv->zv_volsize;
                uint8_t bs = zv->zv_min_bs;

                mutex_exit(&zfsdev_state_lock);
                error = zvol_getefi((void *)arg, flag, vs, bs);
                return (error);
        }

        case DKIOCFLUSHWRITECACHE:
                dkc = (struct dk_callback *)arg;
                mutex_exit(&zfsdev_state_lock);
                zil_commit(zv->zv_zilog, ZVOL_OBJ);
                if ((flag & FKIOCTL) && dkc != NULL && dkc->dkc_callback) {
                        (*dkc->dkc_callback)(dkc->dkc_cookie, error);
                        error = 0;
                }
                return (error);

        case DKIOCGETWCE:
        {
                int wce = (zv->zv_flags & ZVOL_WCE) ? 1 : 0;
                if (ddi_copyout(&wce, (void *)arg, sizeof (int),
                    flag))
                        error = SET_ERROR(EFAULT);
                break;
        }
        case DKIOCSETWCE:
        {
                int wce;
                if (ddi_copyin((void *)arg, &wce, sizeof (int),
                    flag)) {
                        error = SET_ERROR(EFAULT);
                        break;
                }
                if (wce) {
                        zv->zv_flags |= ZVOL_WCE;
                        mutex_exit(&zfsdev_state_lock);
                } else {
                        zv->zv_flags &= ~ZVOL_WCE;
                        mutex_exit(&zfsdev_state_lock);
                        zil_commit(zv->zv_zilog, ZVOL_OBJ);
                }
                return (0);
        }

        case DKIOCGGEOM:
        case DKIOCGVTOC:
```

```
1740                 /*
1741                  * commands using these (like prtvtoc) expect ENOTSUP
1742                  * since we're emulating an EFI label
1743                  */
1744                 error = SET_ERROR(ENOTSUP);
1745                 break;

1747         case DKIOCDUMPINIT:
1748                 rl = zfs_range_lock(&zv->zv_znode, 0, zv->zv_volsize,
1749                     RL_WRITER);
1750                 error = zvol_dumpify(zv);
1751                 zfs_range_unlock(rl);
1752                 break;

1754         case DKIOCDUMPFINI:
1755                 if (!(zv->zv_flags & ZVOL_DUMPIFIED))
1756                         break;
1757                 rl = zfs_range_lock(&zv->zv_znode, 0, zv->zv_volsize,
1758                     RL_WRITER);
1759                 error = zvol_dump_fini(zv);
1760                 zfs_range_unlock(rl);
1761                 break;

1763         case DKIOCFREE:
1764         {
1765                 dkioc_free_t df;
1766                 dmu_tx_t *tx;

1768                 if (ddi_copyin((void *)arg, &df, sizeof (df), flag)) {
1769                         error = SET_ERROR(EFAULT);
1770                         break;
1771                 }

1773                 /*
1774                  * Apply Postel's Law to length-checking.  If they overshoot,
1775                  * just blank out until the end, if there's a need to blank
1776                  * out anything.
1777                  */
1778                 if (df.df_start >= zv->zv_volsize)
1779                         break;  /* No need to do anything... */
1780                 if (df.df_start + df.df_length > zv->zv_volsize)
1781                         df.df_length = DMU_OBJECT_END;

1783                 rl = zfs_range_lock(&zv->zv_znode, df.df_start, df.df_length,
1784                     RL_WRITER);
1785                 tx = dmu_tx_create(zv->zv_objset);
1786                 error = dmu_tx_assign(tx, TXG_WAIT);
1787                 if (error != 0) {
1788                         dmu_tx_abort(tx);
1789                 } else {
1790                         zvol_log_truncate(zv, tx, df.df_start,
1791                             df.df_length, B_TRUE);
1792                         dmu_tx_commit(tx);
1793                         error = dmu_free_long_range(zv->zv_objset, ZVOL_OBJ,
1794                             df.df_start, df.df_length);
1795                 }

1797                 zfs_range_unlock(rl);

1799                 if (error == 0) {
1800                         /*
1801                          * If the write-cache is disabled or 'sync' property
1802                          * is set to 'always' then treat this as a synchronous
1803                          * operation (i.e. commit to zil).
1804                          */
1805                         if (!(zv->zv_flags & ZVOL_WCE) ||
```



```
1806                             (zv->zv_objset->os_sync == ZFS_SYNC_ALWAYS))
1807                                 zil_commit(zv->zv_zilog, ZVOL_OBJ);

1809                         /*
1810                          * If the caller really wants synchronous writes, and
1811                          * can't wait for them, don't return until the write
1812                          * is done.
1813                          */
1814                         if (df.df_flags & DF_WAIT_SYNC) {
1815                                 txg_wait_synced(
1816                                     dmu_objset_pool(zv->zv_objset), 0);
1817                         }
1818                 }
1819                 break;
1820         }

1822         default:
1823                 error = SET_ERROR(ENOTTY);
1824                 break;

1826         }
1827         mutex_exit(&zfsdev_state_lock);
1828         return (error);
1829 }
```

unchanged_portion_omitted