```
**********************************************************
   41410 Thu Aug 27 12:50:06 2015
new/usr/src/lib/libzfs/common/libzfs_import.c
patch v2
6120 libzfs leaks a config nvlist for spares and l2arc
Reviewed by: Igor Kozhukhov <ikozhukhov@gmail.com>
Reviewed by: Toomas Soome <tsoome@me.com>
Reviewed by: Matthew Ahrens <mahrens@delphix.com>
**********************************************************
_____unchanged_portion_omitted_

 211 /*
 212  * Add the given configuration to the list of known devices.
 213  */
 214 static int
 215 add_config(libzfs_handle_t *hdl, pool_list_t *pl, const char *path,
 216     nvlist_t *config)
 217 {
 218         uint64_t pool_guid, vdev_guid, top_guid, txg, state;
 219         pool_entry_t *pe;
 220         vdev_entry_t *ve;
 221         config_entry_t *ce;
 222         name_entry_t *ne;

 224         /*
 225          * If this is a hot spare not currently in use or level 2 cache
 226          * device, add it to the list of names to translate, but don't do
 227          * anything else.
 228          */
 229         if (nvlist_lookup_uint64(config, ZPOOL_CONFIG_POOL_STATE,
 230             &state) == 0 &&
 231             (state == POOL_STATE_SPARE || state == POOL_STATE_L2CACHE) &&
 232             nvlist_lookup_uint64(config, ZPOOL_CONFIG_GUID, &vdev_guid) == 0) {
 233                 if ((ne = zfs_alloc(hdl, sizeof (name_entry_t))) == NULL)
 234                         return (-1);

 236                 if ((ne->ne_name = zfs_strdup(hdl, path)) == NULL) {
 237                         free(ne);
 238                         return (-1);
 239                 }
 240                 ne->ne_guid = vdev_guid;
 241                 ne->ne_next = pl->names;
 242                 pl->names = ne;
 243                 nvlist_free(config);
 244 #endif /* ! codereview */
 245                 return (0);
 246         }

 248         /*
 249          * If we have a valid config but cannot read any of these fields, then
 250          * it means we have a half-initialized label.  In vdev_label_init()
 251          * we write a label with txg == 0 so that we can identify the device
 252          * in case the user refers to the same disk later on.  If we fail to
 253          * create the pool, we'll be left with a label in this state
 254          * which should not be considered part of a valid pool.
 255          */
 256         if (nvlist_lookup_uint64(config, ZPOOL_CONFIG_POOL_GUID,
 257             &pool_guid) != 0 ||
 258             nvlist_lookup_uint64(config, ZPOOL_CONFIG_GUID,
 259             &vdev_guid) != 0 ||
 260             nvlist_lookup_uint64(config, ZPOOL_CONFIG_TOP_GUID,
 261             &top_guid) != 0 ||
 262             nvlist_lookup_uint64(config, ZPOOL_CONFIG_POOL_TXG,
 263             &txg) != 0 || txg == 0) {
 264                 nvlist_free(config);
 265                 return (0);
```



```
 266         }

 268         /*
 269          * First, see if we know about this pool.  If not, then add it to the
 270          * list of known pools.
 271          */
 272         for (pe = pl->pools; pe != NULL; pe = pe->pe_next) {
 273                 if (pe->pe_guid == pool_guid)
 274                         break;
 275         }

 277         if (pe == NULL) {
 278                 if ((pe = zfs_alloc(hdl, sizeof (pool_entry_t))) == NULL)
 243                 if ((pe = zfs_alloc(hdl, sizeof (pool_entry_t))) == NULL) {
 244                         nvlist_free(config);
 279                         return (-1);
 246                 }
 280                 pe->pe_guid = pool_guid;
 281                 pe->pe_next = pl->pools;
 282                 pl->pools = pe;
 283         }

 285         /*
 286          * Second, see if we know about this toplevel vdev.  Add it if its
 287          * missing.
 288          */
 289         for (ve = pe->pe_vdevs; ve != NULL; ve = ve->ve_next) {
 290                 if (ve->ve_guid == top_guid)
 291                         break;
 292         }

 294         if (ve == NULL) {
 295                 if ((ve = zfs_alloc(hdl, sizeof (vdev_entry_t))) == NULL)
 262                 if ((ve = zfs_alloc(hdl, sizeof (vdev_entry_t))) == NULL) {
 263                         nvlist_free(config);
 296                         return (-1);
 265                 }
 297                 ve->ve_guid = top_guid;
 298                 ve->ve_next = pe->pe_vdevs;
 299                 pe->pe_vdevs = ve;
 300         }

 302         /*
 303          * Third, add the vdev guid -> path mappings so that we can fix up
 304          * the configuration as necessary before doing the import.
 305          */
 306         if ((ne = zfs_alloc(hdl, sizeof (name_entry_t))) == NULL)
 307                 return (-1);

 309         if ((ne->ne_name = zfs_strdup(hdl, path)) == NULL) {
 310                 free(ne);
 311                 return (-1);
 312         }

 314         ne->ne_guid = vdev_guid;
 315         ne->ne_next = pl->names;
 316         pl->names = ne;

 318         /*
 319          * Finally, see if we have a config with a matching transaction
 320          * group.  If so, then we do nothing.  Otherwise, add it to the list
 321          * of known configs.
 272          * Third, see if we have a config with a matching transaction group.  If
 273          * so, then we do nothing.  Otherwise, add it to the list of known
 274          * configs.
 322          */
```

```
        for (ce = ve->ve_configs; ce != NULL; ce = ce->ce_next) {
                if (ce->ce_txg == txg)
                        break;
        }

        if (ce == NULL) {
                if ((ce = zfs_alloc(hdl, sizeof (config_entry_t))) == NULL)
                if ((ce = zfs_alloc(hdl, sizeof (config_entry_t))) == NULL) {
                        nvlist_free(config);
                        return (-1);
                }
                ce->ce_txg = txg;
                ce->ce_config = config;
                ce->ce_next = ve->ve_configs;
                ve->ve_configs = ce;
        } else {
                nvlist_free(config);
        }

        /*
         * At this point we've successfully added our config to the list of
         * known configs.  The last thing to do is add the vdev guid -> path
         * mappings so that we can fix up the configuration as necessary before
         * doing the import.
         */
        if ((ne = zfs_alloc(hdl, sizeof (name_entry_t))) == NULL)
                return (-1);

        if ((ne->ne_name = zfs_strdup(hdl, path)) == NULL) {
                free(ne);
                return (-1);
        }

        ne->ne_guid = vdev_guid;
        ne->ne_next = pl->names;
        pl->names = ne;

        return (0);
}
```

unchanged_portion_omitted_

```
/*
 * Given a list of directories to search, find all pools stored on disk.  This
 * includes partial pools which are not available to import.  If no args are
 * given (argc is 0), then the default directory (/dev/dsk) is searched.
 * poolname or guid (but not both) are provided by the caller when trying
 * to import a specific pool.
 */
static nvlist_t *
zpool_find_import_impl(libzfs_handle_t *hdl, importargs_t *iarg)
{
        int i, dirs = iarg->paths;
        struct dirent64 *dp;
        char path[MAXPATHLEN];
        char *end, **dir = iarg->path;
        size_t pathleft;
        nvlist_t *ret = NULL;
        static char *default_dir = "/dev/dsk";
        pool_list_t pools = { 0 };
        pool_entry_t *pe, *penext;
        vdev_entry_t *ve, *venext;
        config_entry_t *ce, *cenext;
        name_entry_t *ne, *nenext;
        avl_tree_t slice_cache;
        rdsk_node_t *slice;
        void *cookie;
```



```
        if (dirs == 0) {
                dirs = 1;
                dir = &default_dir;
        }

        /*
         * Go through and read the label configuration information from every
         * possible device, organizing the information according to pool GUID
         * and toplevel GUID.
         */
        for (i = 0; i < dirs; i++) {
                tpool_t *t;
                char *rdsk;
                int dfd;
                boolean_t config_failed = B_FALSE;
                DIR *dirp;

                /* use realpath to normalize the path */
                if (realpath(dir[i], path) == 0) {
                        (void) zfs_error_fmt(hdl, EZFS_BADPATH,
                            dgettext(TEXT_DOMAIN, "cannot open '%s'"), dir[i]);
                        goto error;
                }
                end = &path[strlen(path)];
                *end++ = '/';
                *end = 0;
                pathleft = &path[sizeof (path)] - end;

                /*
                 * Using raw devices instead of block devices when we're
                 * reading the labels skips a bunch of slow operations during
                 * close(2) processing, so we replace /dev/dsk with /dev/rdsk.
                 */
                if (strcmp(path, "/dev/dsk/") == 0)
                        rdsk = "/dev/rdsk/";
                else
                        rdsk = path;

                if ((dfd = open64(rdsk, O_RDONLY)) < 0 ||
                    (dirp = fdopendir(dfd)) == NULL) {
                        if (dfd >= 0)
                                (void) close(dfd);
                        zfs_error_aux(hdl, strerror(errno));
                        (void) zfs_error_fmt(hdl, EZFS_BADPATH,
                            dgettext(TEXT_DOMAIN, "cannot open '%s'"),
                            rdsk);
                        goto error;
                }

                avl_create(&slice_cache, slice_cache_compare,
                    sizeof (rdsk_node_t), offsetof(rdsk_node_t, rn_node));
                /*
                 * This is not MT-safe, but we have no MT consumers of libzfs
                 */
                while ((dp = readdir64(dirp)) != NULL) {
                        const char *name = dp->d_name;
                        if (name[0] == '.' &&
                            (name[1] == 0 || (name[1] == '.' && name[2] == 0)))
                                continue;

                        slice = zfs_alloc(hdl, sizeof (rdsk_node_t));
                        slice->rn_name = zfs_strdup(hdl, name);
                        slice->rn_avl = &slice_cache;
                        slice->rn_dfd = dfd;
                        slice->rn_hdl = hdl;
```

```
1203                        slice->rn_nozpool = B_FALSE;
1204                        avl_add(&slice_cache, slice);
1205                }
1206                /*
1207                 * create a thread pool to do all of this in parallel;
1208                 * rn_nozpool is not protected, so this is racy in that
1209                 * multiple tasks could decide that the same slice can
1210                 * not hold a zpool, which is benign.  Also choose
1211                 * double the number of processors; we hold a lot of
1212                 * locks in the kernel, so going beyond this doesn't
1213                 * buy us much.
1214                 */
1215                t = tpool_create(1, 2 * sysconf(_SC_NPROCESSORS_ONLN),
1216                    0, NULL);
1217                for (slice = avl_first(&slice_cache); slice;
1218                    (slice = avl_walk(&slice_cache, slice,
1219                    AVL_AFTER)))
1220                        (void) tpool_dispatch(t, zpool_open_func, slice);
1221                tpool_wait(t);
1222                tpool_destroy(t);

1224                cookie = NULL;
1225                while ((slice = avl_destroy_nodes(&slice_cache,
1226                    &cookie)) != NULL) {
1227                        if (slice->rn_config != NULL && !config_failed) {
1228                                nvlist_t *config = slice->rn_config;
1229                                boolean_t matched = B_TRUE;

1231                                if (iarg->poolname != NULL) {
1232                                        char *pname;

1234                                        matched = nvlist_lookup_string(config,
1235                                            ZPOOL_CONFIG_POOL_NAME,
1236                                            &pname) == 0 &&
1237                                            strcmp(iarg->poolname, pname) == 0;
1238                                } else if (iarg->guid != 0) {
1239                                        uint64_t this_guid;

1241                                        matched = nvlist_lookup_uint64(config,
1242                                            ZPOOL_CONFIG_POOL_GUID,
1243                                            &this_guid) == 0 &&
1244                                            iarg->guid == this_guid;
1245                                }
1246                                if (!matched) {
1247                                        nvlist_free(config);
1248                                } else {
1249                                        /*
1250                                         * use the non-raw path for the config
1251                                         */
1252                                        (void) strlcpy(end, slice->rn_name,
1253                                            pathleft);
1254                                        if (add_config(hdl, &pools, path,
1255                                            config) != 0) {
1256                                                nvlist_free(config);
1228                                        config) != 0)
1257                                                config_failed = B_TRUE;
1258                                        }
1259 #endif /* ! codereview */
1260                                }
1261                        }
1262                        free(slice->rn_name);
1263                        free(slice);
1264                }
1265                avl_destroy(&slice_cache);

1267                (void) closedir(dirp);
```



```
1269                if (config_failed)
1270                        goto error;
1271        }

1273        ret = get_configs(hdl, &pools, iarg->can_be_active);

1275 error:
1276        for (pe = pools.pools; pe != NULL; pe = penext) {
1277                penext = pe->pe_next;
1278                for (ve = pe->pe_vdevs; ve != NULL; ve = venext) {
1279                        venext = ve->ve_next;
1280                        for (ce = ve->ve_configs; ce != NULL; ce = cenext) {
1281                                cenext = ce->ce_next;
1282                                if (ce->ce_config)
1283                                        nvlist_free(ce->ce_config);
1284                                free(ce);
1285                        }
1286                        free(ve);
1287                }
1288                free(pe);
1289        }

1291        for (ne = pools.names; ne != NULL; ne = nenext) {
1292                nenext = ne->ne_next;
1293                free(ne->ne_name);
1294                free(ne);
1295        }

1297        return (ret);
1298 }

1300 nvlist_t *
1301 zpool_find_import(libzfs_handle_t *hdl, int argc, char **argv)
1302 {
1303        importargs_t iarg = { 0 };

1305        iarg.paths = argc;
1306        iarg.path = argv;

1308        return (zpool_find_import_impl(hdl, &iarg));
1309 }

1311 /*
1312  * Given a cache file, return the contents as a list of importable pools.
1313  * poolname or guid (but not both) are provided by the caller when trying
1314  * to import a specific pool.
1315  */
1316 nvlist_t *
1317 zpool_find_import_cached(libzfs_handle_t *hdl, const char *cachefile,
1318     char *poolname, uint64_t guid)
1319 {
1320        char *buf;
1321        int fd;
1322        struct stat64 statbuf;
1323        nvlist_t *raw, *src, *dst;
1324        nvlist_t *pools;
1325        nvpair_t *elem;
1326        char *name;
1327        uint64_t this_guid;
1328        boolean_t active;

1330        verify(poolname == NULL || guid == 0);

1332        if ((fd = open(cachefile, O_RDONLY)) < 0) {
1333                zfs_error_aux(hdl, "%s", strerror(errno));
```

```
1334                (void) zfs_error(hdl, EZFS_BADCACHE,
1335                    dgettext(TEXT_DOMAIN, "failed to open cache file"));
1336                return (NULL);
1337        }

1339        if (fstat64(fd, &statbuf) != 0) {
1340                zfs_error_aux(hdl, "%s", strerror(errno));
1341                (void) close(fd);
1342                (void) zfs_error(hdl, EZFS_BADCACHE,
1343                    dgettext(TEXT_DOMAIN, "failed to get size of cache file"));
1344                return (NULL);
1345        }

1347        if ((buf = zfs_alloc(hdl, statbuf.st_size)) == NULL) {
1348                (void) close(fd);
1349                return (NULL);
1350        }

1352        if (read(fd, buf, statbuf.st_size) != statbuf.st_size) {
1353                (void) close(fd);
1354                free(buf);
1355                (void) zfs_error(hdl, EZFS_BADCACHE,
1356                    dgettext(TEXT_DOMAIN,
1357                    "failed to read cache file contents"));
1358                return (NULL);
1359        }

1361        (void) close(fd);

1363        if (nvlist_unpack(buf, statbuf.st_size, &raw, 0) != 0) {
1364                free(buf);
1365                (void) zfs_error(hdl, EZFS_BADCACHE,
1366                    dgettext(TEXT_DOMAIN,
1367                    "invalid or corrupt cache file contents"));
1368                return (NULL);
1369        }

1371        free(buf);

1373        /*
1374         * Go through and get the current state of the pools and refresh their
1375         * state.
1376         */
1377        if (nvlist_alloc(&pools, 0, 0) != 0) {
1378                (void) no_memory(hdl);
1379                nvlist_free(raw);
1380                return (NULL);
1381        }

1383        elem = NULL;
1384        while ((elem = nvlist_next_nvpair(raw, elem)) != NULL) {
1385                src = fnvpair_value_nvlist(elem);

1387                name = fnvlist_lookup_string(src, ZPOOL_CONFIG_POOL_NAME);
1388                if (poolname != NULL && strcmp(poolname, name) != 0)
1389                        continue;

1391                this_guid = fnvlist_lookup_uint64(src, ZPOOL_CONFIG_POOL_GUID);
1392                if (guid != 0 && guid != this_guid)
1393                        continue;

1395                if (pool_active(hdl, name, this_guid, &active) != 0) {
1396                        nvlist_free(raw);
1397                        nvlist_free(pools);
1398                        return (NULL);
1399                }
```



```
1401                if (active)
1402                        continue;

1404                if ((dst = refresh_config(hdl, src)) == NULL) {
1405                        nvlist_free(raw);
1406                        nvlist_free(pools);
1407                        return (NULL);
1408                }

1410                if (nvlist_add_nvlist(pools, nvpair_name(elem), dst) != 0) {
1411                        (void) no_memory(hdl);
1412                        nvlist_free(dst);
1413                        nvlist_free(raw);
1414                        nvlist_free(pools);
1415                        return (NULL);
1416                }
1417                nvlist_free(dst);
1418        }

1420        nvlist_free(raw);
1421        return (pools);
1422 }

1424 static int
1425 name_or_guid_exists(zpool_handle_t *zhp, void *data)
1426 {
1427        importargs_t *import = data;
1428        int found = 0;

1430        if (import->poolname != NULL) {
1431                char *pool_name;

1433                verify(nvlist_lookup_string(zhp->zpool_config,
1434                    ZPOOL_CONFIG_POOL_NAME, &pool_name) == 0);
1435                if (strcmp(pool_name, import->poolname) == 0)
1436                        found = 1;
1437        } else {
1438                uint64_t pool_guid;

1440                verify(nvlist_lookup_uint64(zhp->zpool_config,
1441                    ZPOOL_CONFIG_POOL_GUID, &pool_guid) == 0);
1442                if (pool_guid == import->guid)
1443                        found = 1;
1444        }

1446        zpool_close(zhp);
1447        return (found);
1448 }

1450 nvlist_t *
1451 zpool_search_import(libzfs_handle_t *hdl, importargs_t *import)
1452 {
1453        verify(import->poolname == NULL || import->guid == 0);

1455        if (import->unique)
1456                import->exists = zpool_iter(hdl, name_or_guid_exists, import);

1458        if (import->cachefile != NULL)
1459                return (zpool_find_import_cached(hdl, import->cachefile,
1460                    import->poolname, import->guid));

1462        return (zpool_find_import_impl(hdl, import));
1463 }

1465 boolean_t
```

```
1466 find_guid(nvlist_t *nv, uint64_t guid)
1467 {
1468         uint64_t tmp;
1469         nvlist_t **child;
1470         uint_t c, children;

1472         verify(nvlist_lookup_uint64(nv, ZPOOL_CONFIG_GUID, &tmp) == 0);
1473         if (tmp == guid)
1474                 return (B_TRUE);

1476         if (nvlist_lookup_nvlist_array(nv, ZPOOL_CONFIG_CHILDREN,
1477             &child, &children) == 0) {
1478                 for (c = 0; c < children; c++)
1479                         if (find_guid(child[c], guid))
1480                                 return (B_TRUE);
1481         }

1483         return (B_FALSE);
1484 }

1486 typedef struct aux_cbdata {
1487         const char      *cb_type;
1488         uint64_t        cb_guid;
1489         zpool_handle_t  *cb_zhp;
1490 } aux_cbdata_t;

1492 static int
1493 find_aux(zpool_handle_t *zhp, void *data)
1494 {
1495         aux_cbdata_t *cbp = data;
1496         nvlist_t **list;
1497         uint_t i, count;
1498         uint64_t guid;
1499         nvlist_t *nvroot;

1501         verify(nvlist_lookup_nvlist(zhp->zpool_config, ZPOOL_CONFIG_VDEV_TREE,
1502             &nvroot) == 0);

1504         if (nvlist_lookup_nvlist_array(nvroot, cbp->cb_type,
1505             &list, &count) == 0) {
1506                 for (i = 0; i < count; i++) {
1507                         verify(nvlist_lookup_uint64(list[i],
1508                             ZPOOL_CONFIG_GUID, &guid) == 0);
1509                         if (guid == cbp->cb_guid) {
1510                                 cbp->cb_zhp = zhp;
1511                                 return (1);
1512                         }
1513                 }
1514         }

1516         zpool_close(zhp);
1517         return (0);
1518 }

1520 /*
1521  * Determines if the pool is in use.  If so, it returns true and the state of
1522  * the pool as well as the name of the pool.  Both strings are allocated and
1523  * must be freed by the caller.
1524  */
1525 int
1526 zpool_in_use(libzfs_handle_t *hdl, int fd, pool_state_t *state, char **namestr,
1527     boolean_t *inuse)
1528 {
1529         nvlist_t *config;
1530         char *name;
1531         boolean_t ret;
```



```
1532         uint64_t guid, vdev_guid;
1533         zpool_handle_t *zhp;
1534         nvlist_t *pool_config;
1535         uint64_t stateval, isspare;
1536         aux_cbdata_t cb = { 0 };
1537         boolean_t isactive;

1539         *inuse = B_FALSE;

1541         if (zpool_read_label(fd, &config) != 0) {
1542                 (void) no_memory(hdl);
1543                 return (-1);
1544         }

1546         if (config == NULL)
1547                 return (0);

1549         verify(nvlist_lookup_uint64(config, ZPOOL_CONFIG_POOL_STATE,
1550             &stateval) == 0);
1551         verify(nvlist_lookup_uint64(config, ZPOOL_CONFIG_GUID,
1552             &vdev_guid) == 0);

1554         if (stateval != POOL_STATE_SPARE && stateval != POOL_STATE_L2CACHE) {
1555                 verify(nvlist_lookup_string(config, ZPOOL_CONFIG_POOL_NAME,
1556                     &name) == 0);
1557                 verify(nvlist_lookup_uint64(config, ZPOOL_CONFIG_POOL_GUID,
1558                     &guid) == 0);
1559         }

1561         switch (stateval) {
1562         case POOL_STATE_EXPORTED:
1563                 /*
1564                  * A pool with an exported state may in fact be imported
1565                  * read-only, so check the in-core state to see if it's
1566                  * active and imported read-only.  If it is, set
1567                  * its state to active.
1568                  */
1569                 if (pool_active(hdl, name, guid, &isactive) == 0 && isactive &&
1570                     (zhp = zpool_open_canfail(hdl, name)) != NULL) {
1571                         if (zpool_get_prop_int(zhp, ZPOOL_PROP_READONLY, NULL))
1572                                 stateval = POOL_STATE_ACTIVE;

1574                         /*
1575                          * All we needed the zpool handle for is the
1576                          * readonly prop check.
1577                          */
1578                         zpool_close(zhp);
1579                 }

1581                 ret = B_TRUE;
1582                 break;

1584         case POOL_STATE_ACTIVE:
1585                 /*
1586                  * For an active pool, we have to determine if it's really part
1587                  * of a currently active pool (in which case the pool will exist
1588                  * and the guid will be the same), or whether it's part of an
1589                  * active pool that was disconnected without being explicitly
1590                  * exported.
1591                  */
1592                 if (pool_active(hdl, name, guid, &isactive) != 0) {
1593                         nvlist_free(config);
1594                         return (-1);
1595                 }

1597                 if (isactive) {
```

```
1598                        /*
1599                         * Because the device may have been removed while
1600                         * offlined, we only report it as active if the vdev is
1601                         * still present in the config.  Otherwise, pretend like
1602                         * it's not in use.
1603                         */
1604                        if ((zhp = zpool_open_canfail(hdl, name)) != NULL &&
1605                            (pool_config = zpool_get_config(zhp, NULL))
1606                            != NULL) {
1607                                nvlist_t *nvroot;

1609                                verify(nvlist_lookup_nvlist(pool_config,
1610                                    ZPOOL_CONFIG_VDEV_TREE, &nvroot) == 0);
1611                                ret = find_guid(nvroot, vdev_guid);
1612                        } else {
1613                                ret = B_FALSE;
1614                        }

1616                        /*
1617                         * If this is an active spare within another pool, we
1618                         * treat it like an unused hot spare.  This allows the
1619                         * user to create a pool with a hot spare that currently
1620                         * in use within another pool.  Since we return B_TRUE,
1621                         * libdiskmgt will continue to prevent generic consumers
1622                         * from using the device.
1623                         */
1624                        if (ret && nvlist_lookup_uint64(config,
1625                            ZPOOL_CONFIG_IS_SPARE, &isspare) == 0 && isspare)
1626                                stateval = POOL_STATE_SPARE;

1628                        if (zhp != NULL)
1629                                zpool_close(zhp);
1630                } else {
1631                        stateval = POOL_STATE_POTENTIALLY_ACTIVE;
1632                        ret = B_TRUE;
1633                }
1634                break;

1636        case POOL_STATE_SPARE:
1637                /*
1638                 * For a hot spare, it can be either definitively in use, or
1639                 * potentially active.  To determine if it's in use, we iterate
1640                 * over all pools in the system and search for one with a spare
1641                 * with a matching guid.
1642                 *
1643                 * Due to the shared nature of spares, we don't actually report
1644                 * the potentially active case as in use.  This means the user
1645                 * can freely create pools on the hot spares of exported pools,
1646                 * but to do otherwise makes the resulting code complicated, and
1647                 * we end up having to deal with this case anyway.
1648                 */
1649                cb.cb_zhp = NULL;
1650                cb.cb_guid = vdev_guid;
1651                cb.cb_type = ZPOOL_CONFIG_SPARES;
1652                if (zpool_iter(hdl, find_aux, &cb) == 1) {
1653                        name = (char *)zpool_get_name(cb.cb_zhp);
1654                        ret = B_TRUE;
1655                } else {
1656                        ret = B_FALSE;
1657                }
1658                break;

1660        case POOL_STATE_L2CACHE:

1662                /*
1663                 * Check if any pool is currently using this l2cache device.
```



```
1664                 */
1665                cb.cb_zhp = NULL;
1666                cb.cb_guid = vdev_guid;
1667                cb.cb_type = ZPOOL_CONFIG_L2CACHE;
1668                if (zpool_iter(hdl, find_aux, &cb) == 1) {
1669                        name = (char *)zpool_get_name(cb.cb_zhp);
1670                        ret = B_TRUE;
1671                } else {
1672                        ret = B_FALSE;
1673                }
1674                break;

1676        default:
1677                ret = B_FALSE;
1678        }


1681        if (ret) {
1682                if ((*namestr = zfs_strdup(hdl, name)) == NULL) {
1683                        if (cb.cb_zhp)
1684                                zpool_close(cb.cb_zhp);
1685                        nvlist_free(config);
1686                        return (-1);
1687                }
1688                *state = (pool_state_t)stateval;
1689        }

1691        if (cb.cb_zhp)
1692                zpool_close(cb.cb_zhp);

1694        nvlist_free(config);
1695        *inuse = ret;
1696        return (0);
1697 }
```