

new/usr/src/grub/capability

1

\*\*\*\*\*

1338 Fri Aug 2 17:18:11 2013

new/usr/src/grub/capability

3966 zfs lz4 compression (etc) should have bumped grub capability VERSION

\*\*\*\*\*

```
1 #
2 # CDDL HEADER START
3 #
4 # The contents of this file are subject to the terms of the
5 # Common Development and Distribution License (the "License").
6 # You may not use this file except in compliance with the License.
7 #
8 # You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
9 # or http://www.opensolaris.org/os/licensing.
10 # See the License for the specific language governing permissions
11 # and limitations under the License.
12 #
13 # When distributing Covered Code, include this CDDL HEADER in each
14 # file and include the License file at usr/src/OPENSOLARIS.LICENSE.
15 # If applicable, add the following below this CDDL HEADER, with the
16 # fields enclosed by brackets "[]" replaced with your own identifying
17 # information: Portions Copyright [yyyy] [name of copyright owner]
18 #
19 # CDDL HEADER END
20 #
21 # Copyright (c) 2010, Oracle and/or its affiliates. All rights reserved.
22 # Copyright (c) 2012 by Delphix. All rights reserved.
23 #
24 # This file defines the current capabilities of GRUB over and above that
25 # supported by the standard distribution
26 #
27 # The version field contains the version of the associated GRUB software. The
28 # version is incremented by 1 each time there is a bugfix or enhancement to
29 # GRUB necessitating that the boot blocks be reinstalled for that fix or
30 # enhancement to take effect.
31 # The version field contains the version of the associated GRUB software.
32 # The version is incremented by .1 (minor version number) each time there
33 # is a bugfix or enhancement of GRUB. In addition, the major version number
34 # is bumped up by 1 every time a release boundary is crossed. Thus if in S11
35 # the starting version is 3, in S12 the starting version will be 4.
36 # Note that the first major number in each sequence is a whole integer
37 # i.e. 2.0 is truncated to 2 and 3.0 is truncated to 3.
38 #
39 # VERSION=22
40 # NOTE: Live Upgrade is currently unable to handle decimal fractions (i.e.
41 # minor version numbers) so the version number is being bumped up in
42 # integer increments until Live Upgrade is fixed.
43 #
44 # This file and the associated version are Solaris specific and are
45 # not a part of the open source distribution of GRUB.
46 #
47 # VERSION=21
48 #
49 # dboot
50 # xVM
51 # zfs
52 # findroot
```

```

*****
43642 Fri Aug 2 17:18:11 2013
new/usr/src/grub/grub-0.97/stage2/fsys_zfs.c
3966 zfs lz4 compression (etc) should have bumped grub capability VERSION
*****
_____unchanged_portion_omitted_____

959 /*
960 * List of pool features that the grub implementation of ZFS supports for
961 * read. Note that features that are only required for write do not need
962 * to be listed here since grub opens pools in read-only mode.
963 *
964 * When this list is updated the version number in usr/src/grub/capability
965 * must be incremented to ensure the new grub gets installed.
966 #endif /* ! codereview */
967 */
968 static const char *spa_feature_names[] = {
969     "org.illumos:lz4_compress",
970     NULL
971 };

973 /*
974 * Checks whether the MOS features that are active are supported by this
975 * (GRUB's) implementation of ZFS.
976 *
977 * Return:
978 * 0: Success.
979 * errnum: Failure.
980 */
981 static int
982 check_mos_features(dnode_phys_t *mosmdn, char *stack)
983 {
984     uint64_t objnum;
985     dnode_phys_t *dn;
986     uint8_t error = 0;

988     dn = (dnode_phys_t *)stack;
989     stack += DNODE_SIZE;

991     if ((errnum = dnode_get(mosmdn, DMU_POOL_DIRECTORY_OBJECT,
992         DMU_OT_OBJECT_DIRECTORY, dn, stack)) != 0)
993         return (errnum);

995     /*
996     * Find the object number for 'features_for_read' and retrieve its
997     * corresponding dnode. Note that we don't check features_for_write
998     * because GRUB is not opening the pool for write.
999     */
1000     if ((errnum = zap_lookup(dn, DMU_POOL_FEATURES_FOR_READ, &objnum,
1001         stack)) != 0)
1002         return (errnum);

1004     if ((errnum = dnode_get(mosmdn, objnum, DMU_OTN_ZAP_METADATA,
1005         dn, stack)) != 0)
1006         return (errnum);

1008     return (zap_iterate(dn, check_feature, spa_feature_names, stack));
1009 }

1011 /*
1012 * Given a MOS metadnode, get the metadnode of a given filesystem name (fsname),
1013 * e.g. pool/rootfs, or a given object number (obj), e.g. the object number
1014 * of pool/rootfs.
1015 *
1016 * If no fsname and no obj are given, return the DSL_DIR metadnode.
1017 * If fsname is given, return its metadnode and its matching object number.

```

```

1018 * If only obj is given, return the metadnode for this object number.
1019 *
1020 * Return:
1021 * 0 - success
1022 * errnum - failure
1023 */
1024 static int
1025 get_objset_mdn(dnode_phys_t *mosmdn, char *fsname, uint64_t *obj,
1026     dnode_phys_t *mdn, char *stack)
1027 {
1028     uint64_t objnum, headobj;
1029     char *cname, ch;
1030     blkptr_t *bp;
1031     objset_phys_t *osp;
1032     int issnapshot = 0;
1033     char *snapname;

1035     if (fsname == NULL && obj) {
1036         headobj = *obj;
1037         goto skip;
1038     }

1040     if (errnum = dnode_get(mosmdn, DMU_POOL_DIRECTORY_OBJECT,
1041         DMU_OT_OBJECT_DIRECTORY, mdn, stack))
1042         return (errnum);

1044     if (errnum = zap_lookup(mdn, DMU_POOL_ROOT_DATASET, &objnum,
1045         stack))
1046         return (errnum);

1048     if (errnum = dnode_get(mosmdn, objnum, DMU_OT_DSL_DIR, mdn, stack))
1049         return (errnum);

1051     if (fsname == NULL) {
1052         headobj =
1053             ((dsl_dir_phys_t *)DN_BONUS(mdn))->dd_head_dataset_obj;
1054         goto skip;
1055     }

1057     /* take out the pool name */
1058     while (*fsname && !grub_isspace(*fsname) && *fsname != '/')
1059         fsname++;

1061     while (*fsname && !grub_isspace(*fsname)) {
1062         uint64_t childobj;

1064         while (*fsname == '/')
1065             fsname++;

1067         cname = fsname;
1068         while (*fsname && !grub_isspace(*fsname) && *fsname != '/')
1069             fsname++;
1070         ch = *fsname;
1071         *fsname = 0;

1073         snapname = cname;
1074         while (*snapname && !grub_isspace(*snapname) && *snapname !=
1075             '@')
1076             snapname++;
1077         if (*snapname == '@') {
1078             issnapshot = 1;
1079             *snapname = 0;
1080         }
1081         childobj =
1082             ((dsl_dir_phys_t *)DN_BONUS(mdn))->dd_child_dir_zapobj;
1083         if (errnum = dnode_get(mosmdn, childobj,

```

```

1084         DMU_OT_DSL_DIR_CHILD_MAP, mdn, stack))
1085         return (errnum);

1087         if (zap_lookup(mdn, cname, &objnum, stack))
1088             return (ERR_FILESYSTEM_NOT_FOUND);

1090         if (errnum = dnode_get(mosmdn, objnum, DMU_OT_DSL_DIR,
1091             mdn, stack))
1092             return (errnum);

1094         *fsname = ch;
1095         if (issnapshot)
1096             *snapname = '@';
1097     }
1098     headobj = ((dsl_dir_phys_t *)DN_BONUS(mdn))->dd_head_dataset_obj;
1099     if (obj)
1100         *obj = headobj;

1102 skip:
1103     if (errnum = dnode_get(mosmdn, headobj, DMU_OT_DSL_DATASET, mdn, stack))
1104         return (errnum);
1105     if (issnapshot) {
1106         uint64_t snapobj;

1108         snapobj = ((dsl_dataset_phys_t *)DN_BONUS(mdn))->
1109             ds_snapnames_zapobj;

1111         if (errnum = dnode_get(mosmdn, snapobj,
1112             DMU_OT_DSL_DS_SNAP_MAP, mdn, stack))
1113             return (errnum);
1114         if (zap_lookup(mdn, snapname + 1, &headobj, stack))
1115             return (ERR_FILESYSTEM_NOT_FOUND);
1116         if (errnum = dnode_get(mosmdn, headobj,
1117             DMU_OT_DSL_DATASET, mdn, stack))
1118             return (errnum);
1119         if (obj)
1120             *obj = headobj;
1121     }

1123     bp = &((dsl_dataset_phys_t *)DN_BONUS(mdn))->ds_bp;
1124     osp = (objset_phys_t *)stack;
1125     stack += sizeof(objset_phys_t);
1126     if (errnum = zio_read(bp, osp, stack))
1127         return (errnum);

1129     grub_memmove((char *)mdn, (char *)&osp->os_meta_dnode, DNODE_SIZE);

1131     return (0);
1132 }

1134 /*
1135  * For a given XDR packed nvlist, verify the first 4 bytes and move on.
1136  *
1137  * An XDR packed nvlist is encoded as (comments from nvs_xdr_create) :
1138  *
1139  *   encoding method/host endian      (4 bytes)
1140  *   nvl_version                       (4 bytes)
1141  *   nvl_nvflag                        (4 bytes)
1142  *   encoded nvpairs:
1143  *   encoded size of the nvpair        (4 bytes)
1144  *   decoded size of the nvpair        (4 bytes)
1145  *   name string size                  (4 bytes)
1146  *   name string data                  (sizeof(NV_ALIGN4(string)))
1147  *   data type                         (4 bytes)
1148  *   # of elements in the nvpair       (4 bytes)
1149  *   data

```

```

1150  *   2 zero's for the last nvpair
1151  *   (end of the entire list)        (8 bytes)
1152  *
1153  * Return:
1154  *   0 - success
1155  *   1 - failure
1156  */
1157 static int
1158 nvlist_unpack(char *nvlist, char **out)
1159 {
1160     /* Verify if the 1st and 2nd byte in the nvlist are valid. */
1161     if (nvlist[0] != NV_ENCODE_XDR || nvlist[1] != HOST_ENDIAN)
1162         return (1);

1164     *out = nvlist + 4;
1165     return (0);
1166 }

1168 static char *
1169 nvlist_array(char *nvlist, int index)
1170 {
1171     int i, encode_size;

1173     for (i = 0; i < index; i++) {
1174         /* skip the header, nvl_version, and nvl_nvflag */
1175         nvlist = nvlist + 4 * 2;

1177         while (encode_size = BSWAP_32(*(uint32_t *)nvlist))
1178             nvlist += encode_size; /* goto the next nvpair */

1180         nvlist = nvlist + 4 * 2; /* skip the ending 2 zeros - 8 bytes */
1181     }

1183     return (nvlist);
1184 }

1186 /*
1187  * The nvlist_next_nvpair() function returns a handle to the next nvpair in the
1188  * list following nvpair. If nvpair is NULL, the first pair is returned. If
1189  * nvpair is the last pair in the nvlist, NULL is returned.
1190  */
1191 static char *
1192 nvlist_next_nvpair(char *nvl, char *nvpair)
1193 {
1194     char *cur, *prev;
1195     int encode_size;

1197     if (nvl == NULL)
1198         return (NULL);

1200     if (nvpair == NULL) {
1201         /* skip over nvl_version and nvl_nvflag */
1202         nvpair = nvl + 4 * 2;
1203     } else {
1204         /* skip to the next nvpair */
1205         encode_size = BSWAP_32(*(uint32_t *)nvpair);
1206         nvpair += encode_size;
1207     }

1209     /* 8 bytes of 0 marks the end of the list */
1210     if (*(uint64_t *)nvpair == 0)
1211         return (NULL);

1213     return (nvpair);
1214 }

```

```

1216 /*
1217  * This function returns 0 on success and 1 on failure. On success, a string
1218  * containing the name of nvpair is saved in buf.
1219  */
1220 static int
1221 nvpair_name(char *nvp, char *buf, int buflen)
1222 {
1223     int len;
1224
1225     /* skip over encode/decode size */
1226     nvp += 4 * 2;
1227
1228     len = BSWAP_32(*(uint32_t *)nvp);
1229     if (buflen < len + 1)
1230         return (1);
1231
1232     grub_memmove(buf, nvp + 4, len);
1233     buf[len] = '\0';
1234
1235     return (0);
1236 }
1237
1238 /*
1239  * This function retrieves the value of the nvpair in the form of enumerated
1240  * type data_type_t. This is used to determine the appropriate type to pass to
1241  * nvpair_value().
1242  */
1243 static int
1244 nvpair_type(char *nvp)
1245 {
1246     int name_len, type;
1247
1248     /* skip over encode/decode size */
1249     nvp += 4 * 2;
1250
1251     /* skip over name_len */
1252     name_len = BSWAP_32(*(uint32_t *)nvp);
1253     nvp += 4;
1254
1255     /* skip over name */
1256     nvp = nvp + ((name_len + 3) & ~3); /* align */
1257
1258     type = BSWAP_32(*(uint32_t *)nvp);
1259
1260     return (type);
1261 }
1262
1263 static int
1264 nvpair_value(char *nvp, void *val, int valtype, int *nelmp)
1265 {
1266     int name_len, type, slen;
1267     char *strval = val;
1268     uint64_t *intval = val;
1269
1270     /* skip over encode/decode size */
1271     nvp += 4 * 2;
1272
1273     /* skip over name_len */
1274     name_len = BSWAP_32(*(uint32_t *)nvp);
1275     nvp += 4;
1276
1277     /* skip over name */
1278     nvp = nvp + ((name_len + 3) & ~3); /* align */
1279
1280     /* skip over type */
1281     type = BSWAP_32(*(uint32_t *)nvp);

```

```

1282     nvp += 4;
1283
1284     if (type == valtype) {
1285         int nelm;
1286
1287         nelm = BSWAP_32(*(uint32_t *)nvp);
1288         if (valtype != DATA_TYPE_BOOLEAN && nelm < 1)
1289             return (1);
1290         nvp += 4;
1291
1292         switch (valtype) {
1293             case DATA_TYPE_BOOLEAN:
1294                 return (0);
1295
1296             case DATA_TYPE_STRING:
1297                 slen = BSWAP_32(*(uint32_t *)nvp);
1298                 nvp += 4;
1299                 grub_memmove(strval, nvp, slen);
1300                 strval[slen] = '\0';
1301                 return (0);
1302
1303             case DATA_TYPE_UINT64:
1304                 *intval = BSWAP_64(*(uint64_t *)nvp);
1305                 return (0);
1306
1307             case DATA_TYPE_NVLIST:
1308                 *(void **)val = (void *)nvp;
1309                 return (0);
1310
1311             case DATA_TYPE_NVLIST_ARRAY:
1312                 *(void **)val = (void *)nvp;
1313                 if (nelmp)
1314                     *nelmp = nelm;
1315                 return (0);
1316         }
1317     }
1318
1319     return (1);
1320 }
1321
1322 static int
1323 nvlist_lookup_value(char *nvlist, char *name, void *val, int valtype,
1324                    int *nelmp)
1325 {
1326     char *nvpair;
1327
1328     for (nvpair = nvlist_next_nvpair(nvlist, NULL);
1329          nvpair != NULL;
1330          nvpair = nvlist_next_nvpair(nvlist, nvpair)) {
1331         int name_len = BSWAP_32(*(uint32_t *) (nvpair + 4 * 2));
1332         char *nvp_name = nvpair + 4 * 3;
1333
1334         if ((grub_strcmp(nvp_name, name, name_len) == 0) &&
1335             nvpair_type(nvpair) == valtype) {
1336             return (nvpair_value(nvpair, val, valtype, nelmp));
1337         }
1338     }
1339     return (1);
1340 }
1341
1342 /*
1343  * Check if this vdev is online and is in a good state.
1344  */
1345 static int
1346 vdev_validate(char *nv)
1347 {

```

```

1348     uint64_t ival;

1350     if (nvlist_lookup_value(nv, ZPOOL_CONFIG_OFFLINE, &ival,
1351         DATA_TYPE_UINT64, NULL) == 0 ||
1352         nvlist_lookup_value(nv, ZPOOL_CONFIG_FAULTED, &ival,
1353         DATA_TYPE_UINT64, NULL) == 0 ||
1354         nvlist_lookup_value(nv, ZPOOL_CONFIG_REMOVED, &ival,
1355         DATA_TYPE_UINT64, NULL) == 0)
1356         return (ERR_DEV_VALUES);

1358     return (0);
1359 }

1361 /*
1362  * Get a valid vdev pathname/devid from the boot device.
1363  * The caller should already allocate MAXPATHLEN memory for bootpath and devid.
1364  */
1365 static int
1366 vdev_get_bootpath(char *nv, uint64_t inguid, char *devid, char *bootpath,
1367     int is_spare)
1368 {
1369     char type[16];

1371     if (nvlist_lookup_value(nv, ZPOOL_CONFIG_TYPE, &type, DATA_TYPE_STRING,
1372         NULL))
1373         return (ERR_FSYS_CORRUPT);

1375     if (grub_strcmp(type, VDEV_TYPE_DISK) == 0) {
1376         uint64_t guid;

1378         if (vdev_validate(nv) != 0)
1379             return (ERR_NO_BOOTPATH);

1381         if (nvlist_lookup_value(nv, ZPOOL_CONFIG_GUID,
1382             &guid, DATA_TYPE_UINT64, NULL) != 0)
1383             return (ERR_NO_BOOTPATH);

1385         if (guid != inguid)
1386             return (ERR_NO_BOOTPATH);

1388         /* for a spare vdev, pick the disk labeled with "is_spare" */
1389         if (is_spare) {
1390             uint64_t spare = 0;
1391             (void) nvlist_lookup_value(nv, ZPOOL_CONFIG_IS_SPARE,
1392                 &spare, DATA_TYPE_UINT64, NULL);
1393             if (!spare)
1394                 return (ERR_NO_BOOTPATH);
1395         }

1397         if (nvlist_lookup_value(nv, ZPOOL_CONFIG_PHYS_PATH,
1398             bootpath, DATA_TYPE_STRING, NULL) != 0)
1399             bootpath[0] = '\0';

1401         if (nvlist_lookup_value(nv, ZPOOL_CONFIG_DEVID,
1402             devid, DATA_TYPE_STRING, NULL) != 0)
1403             devid[0] = '\0';

1405         if (grub_strlen(bootpath) >= MAXPATHLEN ||
1406             grub_strlen(devid) >= MAXPATHLEN)
1407             return (ERR_WONT_FIT);

1409         return (0);

1411     } else if (grub_strcmp(type, VDEV_TYPE_MIRROR) == 0 ||
1412         grub_strcmp(type, VDEV_TYPE_REPLACING) == 0 ||
1413         (is_spare = (grub_strcmp(type, VDEV_TYPE_SPARE) == 0))) {

```

```

1414         int nelm, i;
1415         char *child;

1417         if (nvlist_lookup_value(nv, ZPOOL_CONFIG_CHILDREN, &child,
1418             DATA_TYPE_NVLIST_ARRAY, &nelm))
1419             return (ERR_FSYS_CORRUPT);

1421         for (i = 0; i < nelm; i++) {
1422             char *child_i;

1424             child_i = nvlist_array(child, i);
1425             if (vdev_get_bootpath(child_i, inguid, devid,
1426                 bootpath, is_spare) == 0)
1427                 return (0);
1428         }

1429     }

1431     return (ERR_NO_BOOTPATH);
1432 }

1434 /*
1435  * Check the disk label information and retrieve needed vdev name-value pairs.
1436  * Return:
1437  * 0 - success
1438  * ERR_* - failure
1439  */
1440 static int
1441 check_pool_label(uint64_t sector, char *stack, char *outdevid,
1442     char *outpath, uint64_t *outguid, uint64_t *outashift, uint64_t *outversion)
1443 {
1444     vdev_phys_t *vdev;
1445     uint64_t pool_state, txg = 0;
1446     char *nvlist, *nv, *features;
1447     uint64_t diskguid;

1448     sector += (VDEV_SKIP_SIZE >> SPA_MINBLOCKSHIFT);

1452     /* Read in the vdev name-value pair list (112K). */
1453     if (devread(sector, 0, VDEV_PHYS_SIZE, stack) == 0)
1454         return (ERR_READ);

1456     vdev = (vdev_phys_t *)stack;
1457     stack += sizeof (vdev_phys_t);

1459     if (nvlist_unpack(vdev->vp_nvlist, &nvlist))
1460         return (ERR_FSYS_CORRUPT);

1462     if (nvlist_lookup_value(nvlist, ZPOOL_CONFIG_POOL_STATE, &pool_state,
1463         DATA_TYPE_UINT64, NULL))
1464         return (ERR_FSYS_CORRUPT);

1466     if (pool_state == POOL_STATE_DESTROYED)
1467         return (ERR_FILESYSTEM_NOT_FOUND);

1469     if (nvlist_lookup_value(nvlist, ZPOOL_CONFIG_POOL_NAME,
1470         current_rootpool, DATA_TYPE_STRING, NULL))
1471         return (ERR_FSYS_CORRUPT);

1473     if (nvlist_lookup_value(nvlist, ZPOOL_CONFIG_POOL_TXG, &txg,
1474         DATA_TYPE_UINT64, NULL))
1475         return (ERR_FSYS_CORRUPT);

1477     /* not an active device */
1478     if (txg == 0)
1479         return (ERR_NO_BOOTPATH);

```

```

1481     if (nvlist_lookup_value(nvlist, ZPOOL_CONFIG_VERSION, outversion,
1482         DATA_TYPE_UINT64, NULL))
1483         return (ERR_FSYS_CORRUPT);
1484     if (!SPA_VERSION_IS_SUPPORTED(*outversion))
1485         return (ERR_NEWER_VERSION);
1486     if (nvlist_lookup_value(nvlist, ZPOOL_CONFIG_VDEV_TREE, &nv,
1487         DATA_TYPE_NVLIST, NULL))
1488         return (ERR_FSYS_CORRUPT);
1489     if (nvlist_lookup_value(nvlist, ZPOOL_CONFIG_GUID, &diskguid,
1490         DATA_TYPE_UINT64, NULL))
1491         return (ERR_FSYS_CORRUPT);
1492     if (nvlist_lookup_value(nv, ZPOOL_CONFIG_ASHIFT, outashift,
1493         DATA_TYPE_UINT64, NULL) != 0)
1494         return (ERR_FSYS_CORRUPT);
1495     if (vdev_get_bootpath(nv, diskguid, outdevid, outpath, 0))
1496         return (ERR_NO_BOOTPATH);
1497     if (nvlist_lookup_value(nvlist, ZPOOL_CONFIG_POOL_GUID, outguid,
1498         DATA_TYPE_UINT64, NULL))
1499         return (ERR_FSYS_CORRUPT);

1501     if (nvlist_lookup_value(nvlist, ZPOOL_CONFIG_FEATURES_FOR_READ,
1502         &features, DATA_TYPE_NVLIST, NULL) == 0) {
1503         char *nvp;
1504         char *name = stack;
1505         stack += MAXNAMELEN;

1507         for (nvp = nvlist_next_nvpair(features, NULL);
1508             nvp != NULL;
1509             nvp = nvlist_next_nvpair(features, nvp)) {
1510             zap_attribute_t za;

1512             if (nvpair_name(nvp, name, MAXNAMELEN) != 0)
1513                 return (ERR_FSYS_CORRUPT);

1515             za.za_integer_length = 8;
1516             za.za_num_integers = 1;
1517             za.za_first_integer = 1;
1518             za.za_name = name;
1519             if (check_feature(&za, spa_feature_names, stack) != 0)
1520                 return (ERR_NEWER_VERSION);
1521         }
1522     }

1524     return (0);
1525 }

1527 /*
1528  * zfs_mount() locates a valid uberblock of the root pool and read in its MOS
1529  * to the memory address MOS.
1530  *
1531  * Return:
1532  *     1 - success
1533  *     0 - failure
1534  */
1535 int
1536 zfs_mount(void)
1537 {
1538     char *stack, *ub_array;
1539     int label = 0;
1540     uberblock_t *ubbest;
1541     objset_phys_t *osp;
1542     char tmp_bootpath[MAXNAMELEN];
1543     char tmp_devid[MAXNAMELEN];
1544     uint64_t tmp_guid, ashift, version;
1545     uint64_t adjpl = (uint64_t)part_length << SPA_MINBLOCKSHIFT;

```

```

1546     int err = errnum; /* preserve previous errnum state */

1548     /* if it's our first time here, zero the best uberblock out */
1549     if (best_drive == 0 && best_part == 0 && find_best_root) {
1550         grub_memset(&current_uberblock, 0, sizeof (uberblock_t));
1551         pool_guid = 0;
1552     }

1554     stackbase = ZFS_SCRATCH;
1555     stack = stackbase;
1556     ub_array = stack;
1557     stack += VDEV_UBERBLOCK_RING;

1559     osp = (objset_phys_t *)stack;
1560     stack += sizeof (objset_phys_t);
1561     adjpl = P2ALIGN(adjpl, (uint64_t)sizeof (vdev_label_t));

1563     for (label = 0; label < VDEV_LABELS; label++) {

1565         /*
1566          * some eltorito stacks don't give us a size and
1567          * we end up setting the size to MAXUINT, further
1568          * some of these devices stop working once a single
1569          * read past the end has been issued. Checking
1570          * for a maximum part_length and skipping the backup
1571          * labels at the end of the slice/partition/device
1572          * avoids breaking down on such devices.
1573          */
1574         if (part_length == MAXUINT && label == 2)
1575             break;

1577         uint64_t sector = vdev_label_start(adjpl,
1578             label) >> SPA_MINBLOCKSHIFT;

1580         /* Read in the uberblock ring (128K). */
1581         if (devread(sector +
1582             ((VDEV_SKIP_SIZE + VDEV_PHYS_SIZE) >> SPA_MINBLOCKSHIFT),
1583             0, VDEV_UBERBLOCK_RING, ub_array) == 0)
1584             continue;

1586         if (check_pool_label(sector, stack, tmp_devid,
1587             tmp_bootpath, &tmp_guid, &ashift, &version))
1588             continue;

1590         if (pool_guid == 0)
1591             pool_guid = tmp_guid;

1593         if ((ubbest = find_bestub(ub_array, ashift, sector)) == NULL ||
1594             zio_read(&ubbest->ub_rootbp, osp, stack) != 0)
1595             continue;

1597         VERIFY_OS_TYPE(osp, DMU_OST_META);

1599         if (version >= SPA_VERSION_FEATURES &&
1600             check_mos_features(&osp->os_meta_dnode, stack) != 0)
1601             continue;

1603         if (find_best_root && ((pool_guid != tmp_guid) ||
1604             vdev_uberblock_compare(ubbest, &(current_uberblock)) <= 0))
1605             continue;

1607         /* Got the MOS. Save it at the memory addr MOS. */
1608         grub_memmove(MOS, &osp->os_meta_dnode, DNODE_SIZE);
1609         grub_memmove(&current_uberblock, ubbest, sizeof (uberblock_t));
1610         grub_memmove(current_bootpath, tmp_bootpath, MAXNAMELEN);
1611         grub_memmove(current_devid, tmp_devid, grub_strlen(tmp_devid));

```

```

1612         is_zfs_mount = 1;
1613         return (1);
1614     }
1615
1616     /*
1617     * While some fs impls. (tftp) rely on setting and keeping
1618     * global errnums set, others won't reset it and will break
1619     * when issuing rawreads. The goal here is to simply not
1620     * have zfs mount attempts impact the previous state.
1621     */
1622     errnum = err;
1623     return (0);
1624 }
1625
1626 /*
1627 * zfs_open() locates a file in the rootpool by following the
1628 * MOS and places the dnode of the file in the memory address DNODE.
1629 *
1630 * Return:
1631 *     1 - success
1632 *     0 - failure
1633 */
1634 int
1635 zfs_open(char *filename)
1636 {
1637     char *stack;
1638     dnode_phys_t *mdn;
1639
1640     file_buf = NULL;
1641     stackbase = ZFS_SCRATCH;
1642     stack = stackbase;
1643
1644     mdn = (dnode_phys_t *)stack;
1645     stack += sizeof (dnode_phys_t);
1646
1647     dnode_mdn = NULL;
1648     dnode_buf = (dnode_phys_t *)stack;
1649     stack += 1<<DNODE_BLOCK_SHIFT;
1650
1651     /*
1652     * menu.lst is placed at the root pool filesystem level,
1653     * do not goto 'current_bootfs'.
1654     */
1655     if (is_top_dataset_file(filename)) {
1656         if (errnum = get_objset_mdn(MOS, NULL, NULL, mdn, stack))
1657             return (0);
1658
1659         current_bootfs_obj = 0;
1660     } else {
1661         if (current_bootfs[0] == '\0') {
1662             /* Get the default root filesystem object number */
1663             if (errnum = get_default_bootfsobj(MOS,
1664                 &current_bootfs_obj, stack))
1665                 return (0);
1666
1667             if (errnum = get_objset_mdn(MOS, NULL,
1668                 &current_bootfs_obj, mdn, stack))
1669                 return (0);
1670         } else {
1671             if (errnum = get_objset_mdn(MOS, current_bootfs,
1672                 &current_bootfs_obj, mdn, stack)) {
1673                 grub_memset(current_bootfs, 0, MAXNAMELEN);
1674                 return (0);
1675             }
1676         }
1677     }

```

```

1679         if (dnode_get_path(mdn, filename, DNODE, stack)) {
1680             errnum = ERR_FILE_NOT_FOUND;
1681             return (0);
1682         }
1683
1684     /* get the file size and set the file position to 0 */
1685
1686     /*
1687     * For DMU_OT_SA we will need to locate the SIZE attribute
1688     * attribute, which could be either in the bonus buffer
1689     * or the "spill" block.
1690     */
1691     if (DNODE->dn_bonustype == DMU_OT_SA) {
1692         sa_hdr_phys_t *sahdrp;
1693         int hdrsize;
1694
1695         if (DNODE->dn_bonuslen != 0) {
1696             sahdrp = (sa_hdr_phys_t *)DN_BONUS(DNODE);
1697         } else {
1698             if (DNODE->dn_flags & DNODE_FLAG_SPILL_BLKPTR) {
1699                 blkptr_t *bp = &DNODE->dn_spill;
1700                 void *buf;
1701
1702                 buf = (void *)stack;
1703                 stack += BP_GET_LSIZE(bp);
1704
1705                 /* reset errnum to rawread() failure */
1706                 errnum = 0;
1707                 if (zio_read(bp, buf, stack) != 0) {
1708                     return (0);
1709                 }
1710                 sahdrp = buf;
1711             } else {
1712                 errnum = ERR_FSYS_CORRUPT;
1713                 return (0);
1714             }
1715         }
1716         hdrsize = SA_HDR_SIZE(sahdrp);
1717         filemax = *(uint64_t *)((char *)sahdrp + hdrsize +
1718             SA_SIZE_OFFSET);
1719     } else {
1720         filemax = ((znode_phys_t *)DN_BONUS(DNODE))->zp_size;
1721     }
1722     filepos = 0;
1723
1724     dnode_buf = NULL;
1725     return (1);
1726 }
1727
1728 /*
1729 * zfs_read reads in the data blocks pointed by the DNODE.
1730 *
1731 * Return:
1732 *     len - the length successfully read in to the buffer
1733 *     0 - failure
1734 */
1735 int
1736 zfs_read(char *buf, int len)
1737 {
1738     char *stack;
1739     int blksz, length, movesize;
1740
1741     if (file_buf == NULL) {
1742         file_buf = stackbase;
1743         stackbase += SPA_MAXBLOCKSIZE;

```

```
1744     file_start = file_end = 0;
1745 }
1746 stack = stackbase;

1748 /*
1749  * If offset is in memory, move it into the buffer provided and return.
1750  */
1751 if (filepos >= file_start && filepos+len <= file_end) {
1752     grub_memmove(buf, file_buf + filepos - file_start, len);
1753     filepos += len;
1754     return (len);
1755 }

1757 blkosz = DNODE->dn_datablkoszsec << SPA_MINBLOCKSHIFT;

1759 /*
1760  * Entire Dnode is too big to fit into the space available. We
1761  * will need to read it in chunks. This could be optimized to
1762  * read in as large a chunk as there is space available, but for
1763  * now, this only reads in one data block at a time.
1764  */
1765 length = len;
1766 while (length) {
1767     /*
1768      * Find requested blkid and the offset within that block.
1769      */
1770     uint64_t blkid = filepos / blkosz;

1772     if (errno = dmu_read(DNODE, blkid, file_buf, stack))
1773         return (0);

1775     file_start = blkid * blkosz;
1776     file_end = file_start + blkosz;

1778     movesize = MIN(length, file_end - filepos);

1780     grub_memmove(buf, file_buf + filepos - file_start,
1781                 movesize);
1782     buf += movesize;
1783     length -= movesize;
1784     filepos += movesize;
1785 }

1787     return (len);
1788 }

1790 /*
1791  * No-Op
1792  */
1793 int
1794 zfs_embed(int *start_sector, int needed_sectors)
1795 {
1796     return (1);
1797 }

1799 #endif /* FSYS_ZFS */
```