```
**********************************************************
    1338 Fri Aug  2 17:18:11 2013
new/usr/src/grub/capability
3966 zfs lz4 compression (etc) should have bumped grub capability VERSION
**********************************************************
```

```
#
# CDDL HEADER START
#
# The contents of this file are subject to the terms of the
# Common Development and Distribution License (the "License").
# You may not use this file except in compliance with the License.
#
# You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
# or http://www.opensolaris.org/os/licensing.
# See the License for the specific language governing permissions
# and limitations under the License.
#
# When distributing Covered Code, include this CDDL HEADER in each
# file and include the License file at usr/src/OPENSOLARIS.LICENSE.
# If applicable, add the following below this CDDL HEADER, with the
# fields enclosed by brackets "[]" replaced with your own identifying
# information: Portions Copyright [yyyy] [name of copyright owner]
#
# CDDL HEADER END
#
# Copyright (c) 2010, Oracle and/or its affiliates. All rights reserved.
# Copyright (c) 2012 by Delphix. All rights reserved.
#
# This file defines the current capabilities of GRUB over and above that
# supported by the standard distribution
#
# The version field contains the version of the associated GRUB software.  The
# version is incremented by 1 each time there is a bugfix or enhancement to
# GRUB necessitating that the boot blocks be reinstalled for that fix or
# enhancement to take effect.
# The version field contains the version of the associated GRUB software.
# The version is incremented by .1 (minor version number) each time there
# is a bugfix or enhancement of GRUB. In addition, the major version number
# is bumped up by 1 every time a release boundary is crossed. Thus if in S11
# the starting version is 3, in S12 the starting version will be 4.
# Note that the first major number in each sequence is a whole integer
# i.e. 2.0 is truncated to 2 and 3.0 is truncated to 3.
#
VERSION=22
# NOTE: Live Upgrade is currently unable to handle decimal fractions (i.e.
# minor version numbers) so the version number is being bumped up in
# integer increments until Live Upgrade is fixed.
#
# This file and the associated version are Solaris specific and are
# not a part of the open source distribution of GRUB.
#
VERSION=21
dboot
xVM
zfs
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_
```

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

/*
 * Checks whether the MOS features that are active are supported by this
 * (GRUB’s) implementation of ZFS.
 *
 * Return:
 *      0: Success.
 *      errnum: Failure.
 */
static int
check_mos_features(dnode_phys_t *mosmdn, char *stack)
{
        uint64_t objnum;
        dnode_phys_t *dn;
        uint8_t error = 0;

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

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

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

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

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

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

```
 * If only obj is given, return the metadnode for this object number.
 *
 * Return:
 *      0 - success
 *      errnum - failure
 */
static int
get_objset_mdn(dnode_phys_t *mosmdn, char *fsname, uint64_t *obj,
    dnode_phys_t *mdn, char *stack)
{
        uint64_t objnum, headobj;
        char *cname, ch;
        blkptr_t *bp;
        objset_phys_t *osp;
        int issnapshot = 0;
        char *snapname;

        if (fsname == NULL && obj) {
                headobj = *obj;
                goto skip;
        }

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

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

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

        if (fsname == NULL) {
                headobj =
                    ((dsl_dir_phys_t *)DN_BONUS(mdn))->dd_head_dataset_obj;
                goto skip;
        }

        /* take out the pool name */
        while (*fsname && !grub_isspace(*fsname) && *fsname != ’/’)
                fsname++;

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

                while (*fsname == ’/’)
                        fsname++;

                cname = fsname;
                while (*fsname && !grub_isspace(*fsname) && *fsname != ’/’)
                        fsname++;
                ch = *fsname;
                *fsname = 0;

                snapname = cname;
                while (*snapname && !grub_isspace(*snapname) && *snapname !=
                    ’@’)
                        snapname++;
                if (*snapname == ’@’) {
                        issnapshot = 1;
                        *snapname = 0;
                }
                childobj =
                    ((dsl_dir_phys_t *)DN_BONUS(mdn))->dd_child_dir_zapobj;
                if (errnum = dnode_get(mosmdn, childobj,
```

```c
                    DMU_OT_DSL_DIR_CHILD_MAP, mdn, stack))
                        return (errnum);

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

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

                *fsname = ch;
                if (issnapshot)
                        *snapname = '@';
        }
        headobj = ((dsl_dir_phys_t *)DN_BONUS(mdn))->dd_head_dataset_obj;
        if (obj)
                *obj = headobj;

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

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

                if (errnum = dnode_get(mosmdn, snapobj,
                    DMU_OT_DSL_DS_SNAP_MAP, mdn, stack))
                        return (errnum);
                if (zap_lookup(mdn, snapname + 1, &headobj, stack))
                        return (ERR_FILESYSTEM_NOT_FOUND);
                if (errnum = dnode_get(mosmdn, headobj,
                    DMU_OT_DSL_DATASET, mdn, stack))
                        return (errnum);
                if (obj)
                        *obj = headobj;
        }

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

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

        return (0);
}

/*
 * For a given XDR packed nvlist, verify the first 4 bytes and move on.
 *
 * An XDR packed nvlist is encoded as (comments from nvs_xdr_create) :
 *
 *      encoding method/host endian     (4 bytes)
 *      nvl_version                     (4 bytes)
 *      nvl_nvflag                      (4 bytes)
 *      encoded nvpairs:
 *              encoded size of the nvpair      (4 bytes)
 *              decoded size of the nvpair      (4 bytes)
 *              name string size                (4 bytes)
 *              name string data                (sizeof(NV_ALIGN4(string))
 *              data type                       (4 bytes)
 *              # of elements in the nvpair     (4 bytes)
 *              data
```

```c
 *      2 zero's for the last nvpair
 *              (end of the entire list)        (8 bytes)
 *
 * Return:
 *      0 - success
 *      1 - failure
 */
static int
nvlist_unpack(char *nvlist, char **out)
{
        /* Verify if the 1st and 2nd byte in the nvlist are valid. */
        if (nvlist[0] != NV_ENCODE_XDR || nvlist[1] != HOST_ENDIAN)
                return (1);

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

static char *
nvlist_array(char *nvlist, int index)
{
        int i, encode_size;

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

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

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

        return (nvlist);
}

/*
 * The nvlist_next_nvpair() function returns a handle to the next nvpair in the
 * list following nvpair. If nvpair is NULL, the first pair is returned. If
 * nvpair is the last pair in the nvlist, NULL is returned.
 */
static char *
nvlist_next_nvpair(char *nvl, char *nvpair)
{
        char *cur, *prev;
        int encode_size;

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

        if (nvpair == NULL) {
                /* skip over nvl_version and nvl_nvflag */
                nvpair = nvl + 4 * 2;
        } else {
                /* skip to the next nvpair */
                encode_size = BSWAP_32(*(uint32_t *)nvpair);
                nvpair += encode_size;
        }

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

        return (nvpair);
}
```

```c
/*
 * This function returns 0 on success and 1 on failure. On success, a string
 * containing the name of nvpair is saved in buf.
 */
static int
nvpair_name(char *nvp, char *buf, int buflen)
{
        int len;

        /* skip over encode/decode size */
        nvp += 4 * 2;

        len = BSWAP_32(*(uint32_t *)nvp);
        if (buflen < len + 1)
                return (1);

        grub_memmove(buf, nvp + 4, len);
        buf[len] = '\0';

        return (0);
}

/*
 * This function retrieves the value of the nvpair in the form of enumerated
 * type data_type_t. This is used to determine the appropriate type to pass to
 * nvpair_value().
 */
static int
nvpair_type(char *nvp)
{
        int name_len, type;

        /* skip over encode/decode size */
        nvp += 4 * 2;

        /* skip over name_len */
        name_len = BSWAP_32(*(uint32_t *)nvp);
        nvp += 4;

        /* skip over name */
        nvp = nvp + ((name_len + 3) & ~3); /* align */

        type = BSWAP_32(*(uint32_t *)nvp);

        return (type);
}

static int
nvpair_value(char *nvp, void *val, int valtype, int *nelmp)
{
        int name_len, type, slen;
        char *strval = val;
        uint64_t *intval = val;

        /* skip over encode/decode size */
        nvp += 4 * 2;

        /* skip over name_len */
        name_len = BSWAP_32(*(uint32_t *)nvp);
        nvp += 4;

        /* skip over name */
        nvp = nvp + ((name_len + 3) & ~3); /* align */

        /* skip over type */
        type = BSWAP_32(*(uint32_t *)nvp);
```

```c
        nvp += 4;

        if (type == valtype) {
                int nelm;

                nelm = BSWAP_32(*(uint32_t *)nvp);
                if (valtype != DATA_TYPE_BOOLEAN && nelm < 1)
                        return (1);
                nvp += 4;

                switch (valtype) {
                case DATA_TYPE_BOOLEAN:
                        return (0);

                case DATA_TYPE_STRING:
                        slen = BSWAP_32(*(uint32_t *)nvp);
                        nvp += 4;
                        grub_memmove(strval, nvp, slen);
                        strval[slen] = '\0';
                        return (0);

                case DATA_TYPE_UINT64:
                        *intval = BSWAP_64(*(uint64_t *)nvp);
                        return (0);

                case DATA_TYPE_NVLIST:
                        *(void **)val = (void *)nvp;
                        return (0);

                case DATA_TYPE_NVLIST_ARRAY:
                        *(void **)val = (void *)nvp;
                        if (nelmp)
                                *nelmp = nelm;
                        return (0);
                }
        }

        return (1);
}

static int
nvlist_lookup_value(char *nvlist, char *name, void *val, int valtype,
    int *nelmp)
{
        char *nvpair;

        for (nvpair = nvlist_next_nvpair(nvlist, NULL);
            nvpair != NULL;
            nvpair = nvlist_next_nvpair(nvlist, nvpair)) {
                int name_len = BSWAP_32(*(uint32_t *)(nvpair + 4 * 2));
                char *nvp_name = nvpair + 4 * 3;

                if ((grub_strncmp(nvp_name, name, name_len) == 0) &&
                    nvpair_type(nvpair) == valtype) {
                        return (nvpair_value(nvpair, val, valtype, nelmp));
                }
        }
        return (1);
}

/*
 * Check if this vdev is online and is in a good state.
 */
static int
vdev_validate(char *nv)
{
        uint64_t ival;

        if (nvlist_lookup_value(nv, ZPOOL_CONFIG_OFFLINE, &ival,
            DATA_TYPE_UINT64, NULL) == 0 ||
            nvlist_lookup_value(nv, ZPOOL_CONFIG_FAULTED, &ival,
            DATA_TYPE_UINT64, NULL) == 0 ||
            nvlist_lookup_value(nv, ZPOOL_CONFIG_REMOVED, &ival,
            DATA_TYPE_UINT64, NULL) == 0)
                return (ERR_DEV_VALUES);

        return (0);
}

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

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

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

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

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

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

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

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

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

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

                return (0);

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

```
                int nelm, i;
                char *child;

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

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

                        child_i = nvlist_array(child, i);
                        if (vdev_get_bootpath(child_i, inguid, devid,
                            bootpath, is_spare) == 0)
                                return (0);
                }
        }

        return (ERR_NO_BOOTPATH);
}

/*
 * Check the disk label information and retrieve needed vdev name-value pairs.
 *
 * Return:
 *      0 - success
 *      ERR_* - failure
 */
static int
check_pool_label(uint64_t sector, char *stack, char *outdevid,
    char *outpath, uint64_t *outguid, uint64_t *outashift, uint64_t *outversion)
{
        vdev_phys_t *vdev;
        uint64_t pool_state, txg = 0;
        char *nvlist, *nv, *features;
        uint64_t diskguid;

        sector += (VDEV_SKIP_SIZE >> SPA_MINBLOCKSHIFT);

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

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

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

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

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

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

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

        /* not an active device */
        if (txg == 0)
                return (ERR_NO_BOOTPATH);
        if (nvlist_lookup_value(nvlist, ZPOOL_CONFIG_VERSION, outversion,
            DATA_TYPE_UINT64, NULL))
                return (ERR_FSYS_CORRUPT);
        if (!SPA_VERSION_IS_SUPPORTED(*outversion))
                return (ERR_NEWER_VERSION);
        if (nvlist_lookup_value(nvlist, ZPOOL_CONFIG_VDEV_TREE, &nv,
            DATA_TYPE_NVLIST, NULL))
                return (ERR_FSYS_CORRUPT);
        if (nvlist_lookup_value(nvlist, ZPOOL_CONFIG_GUID, &diskguid,
            DATA_TYPE_UINT64, NULL))
                return (ERR_FSYS_CORRUPT);
        if (nvlist_lookup_value(nv, ZPOOL_CONFIG_ASHIFT, outashift,
            DATA_TYPE_UINT64, NULL) != 0)
                return (ERR_FSYS_CORRUPT);
        if (vdev_get_bootpath(nv, diskguid, outdevid, outpath, 0))
                return (ERR_NO_BOOTPATH);
        if (nvlist_lookup_value(nvlist, ZPOOL_CONFIG_POOL_GUID, outguid,
            DATA_TYPE_UINT64, NULL))
                return (ERR_FSYS_CORRUPT);

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

                for (nvp = nvlist_next_nvpair(features, NULL);
                    nvp != NULL;
                    nvp = nvlist_next_nvpair(features, nvp)) {
                        zap_attribute_t za;

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

                        za.za_integer_length = 8;
                        za.za_num_integers = 1;
                        za.za_first_integer = 1;
                        za.za_name = name;
                        if (check_feature(&za, spa_feature_names, stack) != 0)
                                return (ERR_NEWER_VERSION);
                }
        }

        return (0);
}

/*
 * zfs_mount() locates a valid uberblock of the root pool and read in its MOS
 * to the memory address MOS.
 *
 * Return:
 *      1 - success
 *      0 - failure
 */
int
zfs_mount(void)
{
        char *stack, *ub_array;
        int label = 0;
        uberblock_t *ubbest;
        objset_phys_t *osp;
        char tmp_bootpath[MAXNAMELEN];
        char tmp_devid[MAXNAMELEN];
        uint64_t tmp_guid, ashift, version;
        uint64_t adjpl = (uint64_t)part_length << SPA_MINBLOCKSHIFT;
```

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

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

        stackbase = ZFS_SCRATCH;
        stack = stackbase;
        ub_array = stack;
        stack += VDEV_UBERBLOCK_RING;

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

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

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

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

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

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

                if (pool_guid == 0)
                        pool_guid = tmp_guid;

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

                VERIFY_OS_TYPE(osp, DMU_OST_META);

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

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

                /* Got the MOS. Save it at the memory addr MOS. */
                grub_memmove(MOS, &osp->os_meta_dnode, DNODE_SIZE);
                grub_memmove(&current_uberblock, ubbest, sizeof (uberblock_t));
                grub_memmove(current_bootpath, tmp_bootpath, MAXNAMELEN);
                grub_memmove(current_devid, tmp_devid, grub_strlen(tmp_devid));
                is_zfs_mount = 1;
                return (1);
        }

        /*
         * While some fs impls. (tftp) rely on setting and keeping
         * global errnums set, others won't reset it and will break
         * when issuing rawreads. The goal here is to simply not
         * have zfs mount attempts impact the previous state.
         */
        errnum = err;
        return (0);
}

/*
 * zfs_open() locates a file in the rootpool by following the
 * MOS and places the dnode of the file in the memory address DNODE.
 *
 * Return:
 *      1 - success
 *      0 - failure
 */
int
zfs_open(char *filename)
{
        char *stack;
        dnode_phys_t *mdn;

        file_buf = NULL;
        stackbase = ZFS_SCRATCH;
        stack = stackbase;

        mdn = (dnode_phys_t *)stack;
        stack += sizeof (dnode_phys_t);

        dnode_mdn = NULL;
        dnode_buf = (dnode_phys_t *)stack;
        stack += 1<<DNODE_BLOCK_SHIFT;

        /*
         * menu.lst is placed at the root pool filesystem level,
         * do not goto 'current_bootfs'.
         */
        if (is_top_dataset_file(filename)) {
                if (errnum = get_objset_mdn(MOS, NULL, NULL, mdn, stack))
                        return (0);

                current_bootfs_obj = 0;
        } else {
                if (current_bootfs[0] == '\0') {
                        /* Get the default root filesystem object number */
                        if (errnum = get_default_bootfsobj(MOS,
                            &current_bootfs_obj, stack))
                                return (0);

                        if (errnum = get_objset_mdn(MOS, NULL,
                            &current_bootfs_obj, mdn, stack))
                                return (0);
                } else {
                        if (errnum = get_objset_mdn(MOS, current_bootfs,
                            &current_bootfs_obj, mdn, stack)) {
                                grub_memset(current_bootfs, 0, MAXNAMELEN);
                                return (0);
                        }
                }
        }
```

```
        if (dnode_get_path(mdn, filename, DNODE, stack)) {
                errnum = ERR_FILE_NOT_FOUND;
                return (0);
        }

        /* get the file size and set the file position to 0 */

        /*
         * For DMU_OT_SA we will need to locate the SIZE attribute
         * attribute, which could be either in the bonus buffer
         * or the "spill" block.
         */
        if (DNODE->dn_bonustype == DMU_OT_SA) {
                sa_hdr_phys_t *sahdrp;
                int hdrsize;

                if (DNODE->dn_bonuslen != 0) {
                        sahdrp = (sa_hdr_phys_t *)DN_BONUS(DNODE);
                } else {
                        if (DNODE->dn_flags & DNODE_FLAG_SPILL_BLKPTR) {
                                blkptr_t *bp = &DNODE->dn_spill;
                                void *buf;

                                buf = (void *)stack;
                                stack += BP_GET_LSIZE(bp);

                                /* reset errnum to rawread() failure */
                                errnum = 0;
                                if (zio_read(bp, buf, stack) != 0) {
                                        return (0);
                                }
                                sahdrp = buf;
                        } else {
                                errnum = ERR_FSYS_CORRUPT;
                                return (0);
                        }
                }
                hdrsize = SA_HDR_SIZE(sahdrp);
                filemax = *(uint64_t *)((char *)sahdrp + hdrsize +
                    SA_SIZE_OFFSET);
        } else {
                filemax = ((znode_phys_t *)DN_BONUS(DNODE))->zp_size;
        }
        filepos = 0;

        dnode_buf = NULL;
        return (1);
}

/*
 * zfs_read reads in the data blocks pointed by the DNODE.
 *
 * Return:
 *      len - the length successfully read in to the buffer
 *      0   - failure
 */
int
zfs_read(char *buf, int len)
{
        char *stack;
        int blksz, length, movesize;

        if (file_buf == NULL) {
                file_buf = stackbase;
                stackbase += SPA_MAXBLOCKSIZE;
                file_start = file_end = 0;
        }
        stack = stackbase;

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

        blksz = DNODE->dn_datablkszsec << SPA_MINBLOCKSHIFT;

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

                if (errnum = dmu_read(DNODE, blkid, file_buf, stack))
                        return (0);

                file_start = blkid * blksz;
                file_end = file_start + blksz;

                movesize = MIN(length, file_end - filepos);

                grub_memmove(buf, file_buf + filepos - file_start,
                    movesize);
                buf += movesize;
                length -= movesize;
                filepos += movesize;
        }

        return (len);
}

/*
 * No-Op
 */
int
zfs_embed(int *start_sector, int needed_sectors)
{
        return (1);
}

#endif /* FSYS_ZFS */
```