```
**********************************************************
  228349 Fi Aug 31 05:08:45 2012
new/bootadm/bootadm.c
botadm patch
**********************************************************
```

```
/*
 * 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) 2005, 2010, Oracle and/or its affiliates. All rights reserved.
 * Copyright 2012 Milan Jurik. All rights reserved.
 * Copyright 2012 Daniil Lunev. All rights reserved.
#endif /* ! codereview */
 */

/*
 * Copyright 2011 Nexenta Systems, Inc. All rights reserved.
 */

/*
 * bootadm(1M) is a new utility for managing bootability of
 * Solaris *Newboot* environments. It has two primary tasks:
 *      - Allow end users to manage bootability of Newboot Solaris instances
 *      - Provide services to other subsystems in Solaris (primarily Install)
 */

/* Headers */
#include <stdio.h>
#include <errno.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <alloca.h>
#include <stdarg.h>
#include <limits.h>
#include <signal.h>
#include <sys/wait.h>
#include <sys/mnttab.h>
#include <sys/mntent.h>
#include <sys/statvfs.h>
#include <libnvpair.h>
#include <ftw.h>
#include <fcntl.h>
#include <strings.h>
#include <utime.h>
#include <sys/systeminfo.h>
#include <sys/dktp/fdisk.h>
```



```
#include <sys/param.h>
#include <dirent.h>
#include <ctype.h>
#include <libgen.h>
#include <sys/sysmacros.h>
#include <sys/elf.h>
#include <libscf.h>
#include <zlib.h>
#include <sys/lockfs.h>
#include <sys/filio.h>
#include <libbe.h>
#ifdef i386
#include <libfdisk.h>
#endif

#if !defined(_OPB)
#include <sys/ucode.h>
#endif

#include <pwd.h>
#include <grp.h>
#include <device_info.h>
#include <sys/vtoc.h>
#include <sys/efi_partition.h>
#include <regex.h>
#include <locale.h>

#include "message.h"
#include "bootadm.h"

#ifndef TEXT_DOMAIN
#define TEXT_DOMAIN     "SUNW_OST_OSCMD"
#endif  /* TEXT_DOMAIN */

/* Type definitions */

/* Primary subcmds */
typedef enum {
        BAM_MENU = 3,
        BAM_ARCHIVE
} subcmd_t;

typedef enum {
    OPT_ABSENT = 0,     /* No option */
    OPT_REQ,            /* option required */
    OPT_OPTIONAL        /* option may or may not be present */
} option_t;

typedef struct {
        char    *subcmd;
        option_t option;
        error_t (*handler)();
        int     unpriv;                 /* is this an unprivileged command */
} subcmd_defn_t;

#define LINE_INIT       0       /* lineNum initial value */
#define ENTRY_INIT      -1      /* entryNum initial value */
#define ALL_ENTRIES     -2      /* selects all boot entries */

#define GRUB_DIR                "/boot/grub"
#define GRUB_STAGE2             GRUB_DIR "/stage2"
#define GRUB_MENU               "/boot/illumos.cfg"
#define MENU_TMP                "/boot/illumos.cfg.tmp"
#define GRUB_MENU               "/boot/grub/menu.lst"
#define MENU_TMP                "/boot/grub/menu.lst.tmp"
#define GRUB_BACKUP_MENU        "/etc/lu/GRUB_backup_menu"
```

```
126 #define RAMDISK_SPECIAL         "/ramdisk"
127 #define STUBBOOT                "/stubboot"
128 #define MULTIBOOT               "/platform/i86pc/multiboot"
129 #define GRUBSIGN_DIR            "/boot/grub/bootsign"
130 #define GRUBSIGN_BACKUP         "/etc/bootsign"
131 #define GRUBSIGN_UFS_PREFIX     "rootfs"
132 #define GRUBSIGN_ZFS_PREFIX     "pool_"
133 #define GRUBSIGN_LU_PREFIX      "BE_"
134 #define UFS_SIGNATURE_LIST      "/var/run/grub_ufs_signatures"
135 #define ZFS_LEGACY_MNTPT        "/tmp/bootadm_mnt_zfs_legacy"

137 #define BOOTADM_RDONLY_TEST     "BOOTADM_RDONLY_TEST"

139 /* lock related */
140 #define BAM_LOCK_FILE           "/var/run/bootadm.lock"
141 #define LOCK_FILE_PERMS         (S_IRUSR|S_IWUSR|S_IRGRP|S_IROTH)

143 #define CREATE_RAMDISK          "boot/solaris/bin/create_ramdisk"
144 #define CREATE_DISKMAP          "boot/solaris/bin/create_diskmap"
145 #define EXTRACT_BOOT_FILELIST   "boot/solaris/bin/extract_boot_filelist"
146 #define GRUBDISK_MAP            "/var/run/solaris_grubdisk.map"

148 #define GRUB_slice              "/etc/lu/GRUB_slice"
149 #define GRUB_root               "/etc/lu/GRUB_root"
150 #define GRUB_fdisk              "/etc/lu/GRUB_fdisk"
151 #define GRUB_fdisk_target       "/etc/lu/GRUB_fdisk_target"
152 #define FINDROOT_INSTALLGRUB    "/etc/lu/installgrub.findroot"
153 #define LULIB                   "/usr/lib/lu/lulib"
154 #define LULIB_PROPAGATE_FILE    "lulib_propagate_file"
155 #define CKSUM                   "/usr/bin/cksum"
156 #define LU_MENU_CKSUM           "/etc/lu/menu.cksum"
157 #define BOOTADM                 "/sbin/bootadm"

159 #define INSTALLGRUB             "/sbin/installgrub"
160 #define STAGE1                  "/boot/grub/stage1"
161 #define STAGE2                  "/boot/grub/stage2"

163 typedef enum zfs_mnted {
164         ZFS_MNT_ERROR = -1,
165         LEGACY_MOUNTED = 1,
166         LEGACY_ALREADY,
167         ZFS_MOUNTED,
168         ZFS_ALREADY
169 } zfs_mnted_t;

171 /*
172  * Default file attributes
173  */
174 #define DEFAULT_DEV_MODE        0644    /* default permissions */
175 #define DEFAULT_DEV_UID         0       /* user root */
176 #define DEFAULT_DEV_GID         3       /* group sys */

178 /*
179  * Menu related
180  * menu_cmd_t and menu_cmds must be kept in sync
181  */
182 char *menu_cmds[] = {
183         "default_entry",/* DEFAULT_CMD */
 84         "default",      /* DEFAULT_CMD */
184         "timeout",      /* TIMEOUT_CMD */
185         "entry_name",   /* TITLE_CMD */
186         "pool_uuid",    /* ROOT_CMD */
187         "kernel_path$", /* KERNEL_CMD */
188         "kernel_path",  /* KERNEL_DOLLAR_CMD */
189         "module$",      /* MODULE_CMD */
190         "module",       /* MODULE_DOLLAR_CMD */
```



```
191         "=",            /* SEP_CMD */
 86         "title",        /* TITLE_CMD */
 87         "root",         /* ROOT_CMD */
 88         "kernel",       /* KERNEL_CMD */
 89         "kernel$",      /* KERNEL_DOLLAR_CMD */
 90         "module",       /* MODULE_CMD */
 91         "module$",      /* MODULE_DOLLAR_CMD */
 92         " ",            /* SEP_CMD */
192         "#",            /* COMMENT_CMD */
193         "chainloader",  /* CHAINLOADER_CMD */
194         "args",         /* ARGS_CMD */
195         "pool_label",   /* FINDROOT_CMD */
196         "data_set",     /* BOOTFS_CMD */
197         "kernel_options",/* KERNEL_OPTIONS_CMD */
 96         "findroot",     /* FINDROOT_CMD */
 97         "bootfs",       /* BOOTFS_CMD */
198         NULL
199 };
```

*unchanged_portion_omitted_*

```
4681 int
4682 add_boot_entry(menu_t *mp,
4683         char *title,
4684         char *findroot,
4685         char *kernel,
4686         char *mod_kernel,
4687         char *module,
4688         char *bootfs)
4689 {
4690         int             lineNum;
4691         int             entryNum;
4692         char            linebuf[BAM_MAXLINE];
4693         menu_cmd_t      k_cmd;
4694         menu_cmd_t      m_cmd;
4695         const char      *fcn = "add_boot_entry()";
4696         char * options = NULL;
4697 #endif /* ! codereview */

4699         assert(mp);

4701         INJECT_ERROR1("ADD_BOOT_ENTRY_FINDROOT_NULL", findroot = NULL);
4702         if (findroot == NULL) {
4703                 bam_error(NULL_FINDROOT);
4704                 return (BAM_ERROR);
4705         }

4707         if (title == NULL) {
4708                 title = "Solaris";      /* default to Solaris */
4709         }
4710         if (kernel == NULL) {
4711                 bam_error(SUBOPT_MISS, menu_cmds[KERNEL_CMD]);
4712                 return (BAM_ERROR);
4713         }
4714         if (module == NULL) {
4715                 if (bam_direct != BAM_DIRECT_DBOOT) {
4716                         bam_error(SUBOPT_MISS, menu_cmds[MODULE_CMD]);
4717                         return (BAM_ERROR);
4718                 }

4720                 /* Figure the commands out from the kernel line */
4721                 if (strstr(kernel, "$ISADIR") != NULL) {
4722                         module = DIRECT_BOOT_ARCHIVE;
4723                 } else if (strstr(kernel, "amd64") != NULL) {
4724                         module = DIRECT_BOOT_ARCHIVE_64;
4725                 } else {
4726                         module = DIRECT_BOOT_ARCHIVE_32;
```

```
4727                }
4728        }

4730        k_cmd = KERNEL_DOLLAR_CMD;
4731        m_cmd = MODULE_DOLLAR_CMD;

4733        if (mp->start) {
4734                lineNum = mp->end->lineNum;
4735                entryNum = mp->end->entryNum;
4736        } else {
4737                lineNum = LINE_INIT;
4738                entryNum = ENTRY_INIT;
4739        }

4741        /*
4742         * No separator for comment (HDR/FTR) commands
4743         * The syntax for comments is #<comment>
4744         */
4745        (void) snprintf(linebuf, sizeof (linebuf), "%s%s",
4746            menu_cmds[COMMENT_CMD], BAM_BOOTADM_HDR);
4747        line_parser(mp, linebuf, &lineNum, &entryNum);

4749        (void) snprintf(linebuf, sizeof (linebuf), "%s%s%s",
4750            menu_cmds[TITLE_CMD], menu_cmds[SEP_CMD], title);
4751        line_parser(mp, linebuf, &lineNum, &entryNum);

4753        (void) snprintf(linebuf, sizeof (linebuf), "%s%s%s",
4754            menu_cmds[FINDROOT_CMD], menu_cmds[SEP_CMD], findroot);
4755        line_parser(mp, linebuf, &lineNum, &entryNum);
4756        BAM_DPRINTF((D_ADD_FINDROOT_NUM, fcn, lineNum, entryNum));

4758        if (bootfs != NULL) {
4759                (void) snprintf(linebuf, sizeof (linebuf), "%s%s%s",
4760                    menu_cmds[BOOTFS_CMD], menu_cmds[SEP_CMD], bootfs);
4761                line_parser(mp, linebuf, &lineNum, &entryNum);
4762        }

4764        options = strpbrk(kernel, " \t");
4765        if (options)
4766                ++options;

4768        (void) snprintf(linebuf, sizeof (linebuf), "%s%s",
4769            menu_cmds[k_cmd], menu_cmds[SEP_CMD]);
4770        (void) strncat(linebuf, kernel, options - kernel);
4771        line_parser(mp, linebuf, &lineNum, &entryNum);

4773        if (options) {
4774 #endif /* ! codereview */
4775                (void) snprintf(linebuf, sizeof (linebuf), "%s%s%s",
4776                    menu_cmds[KERNEL_OPTIONS_CMD], menu_cmds[SEP_CMD], options);
4596            menu_cmds[k_cmd], menu_cmds[SEP_CMD], kernel);
4777                line_parser(mp, linebuf, &lineNum, &entryNum);
4778        }
4779 #endif /* ! codereview */

4781        if (mod_kernel != NULL) {
4782                (void) snprintf(linebuf, sizeof (linebuf), "%s%s%s",
4783                    menu_cmds[m_cmd], menu_cmds[SEP_CMD], mod_kernel);
4784                line_parser(mp, linebuf, &lineNum, &entryNum);
4785        }

4787        (void) snprintf(linebuf, sizeof (linebuf), "%s%s%s",
4788            menu_cmds[m_cmd], menu_cmds[SEP_CMD], module);
4789        line_parser(mp, linebuf, &lineNum, &entryNum);

4791        (void) snprintf(linebuf, sizeof (linebuf), "%s%s",
```



```
4792            menu_cmds[COMMENT_CMD], BAM_BOOTADM_FTR);
4793        line_parser(mp, linebuf, &lineNum, &entryNum);

4795        return (entryNum);
4796 }

4798 error_t
4799 delete_boot_entry(menu_t *mp, int entryNum, int quiet)
4800 {
4801        line_t          *lp;
4802        line_t          *freed;
4803        entry_t         *ent;
4804        entry_t         *tmp;
4805        int             deleted = 0;
4806        const char      *fcn = "delete_boot_entry()";

4808        assert(entryNum != ENTRY_INIT);

4810        tmp = NULL;

4812        ent = mp->entries;
4813        while (ent) {
4814                lp = ent->start;

4816                /*
4817                 * Check entry number and make sure it's a modifiable entry.
4818                 *
4819                 * Guidelines:
4820                 *      + We can modify a bootadm-created entry
4821                 *      + We can modify a libbe-created entry
4822                 */
4823                if ((lp->flags != BAM_COMMENT &&
4824                    (((ent->flags & BAM_ENTRY_LIBBE) == 0) &&
4825                    strcmp(lp->arg, BAM_BOOTADM_HDR) != 0)) ||
4826                    (entryNum != ALL_ENTRIES && lp->entryNum != entryNum)) {
4827                        ent = ent->next;
4828                        continue;
4829                }

4831                /* free the entry content */
4832                do {
4833                        freed = lp;
4834                        lp = lp->next;  /* prev stays the same */
4835                        BAM_DPRINTF((D_FREEING_LINE, fcn, freed->lineNum));
4836                        unlink_line(mp, freed);
4837                        line_free(freed);
4838                } while (freed != ent->end);

4840                /* free the entry_t structure */
4841                assert(tmp == NULL);
4842                tmp = ent;
4843                ent = ent->next;
4844                if (tmp->prev)
4845                        tmp->prev->next = ent;
4846                else
4847                        mp->entries = ent;
4848                if (ent)
4849                        ent->prev = tmp->prev;
4850                BAM_DPRINTF((D_FREEING_ENTRY, fcn, tmp->entryNum));
4851                free(tmp);
4852                tmp = NULL;
4853                deleted = 1;
4854        }

4856        assert(tmp == NULL);
```

```
4858         if (!deleted && entryNum != ALL_ENTRIES) {
4859                 if (quiet == DBE_PRINTERR)
4860                         bam_error(NO_BOOTADM_MATCH);
4861                 return (BAM_ERROR);
4862         }
4863
4864         /*
4865          * Now that we have deleted an entry, update
4866          * the entry numbering and the default cmd.
4867          */
4868         update_numbering(mp);
4869
4870         return (BAM_SUCCESS);
4871 }
4872
4873 static error_t
4874 delete_all_entries(menu_t *mp, char *dummy, char *opt)
4875 {
4876         assert(mp);
4877         assert(dummy == NULL);
4878         assert(opt == NULL);
4879
4880         BAM_DPRINTF((D_FUNC_ENTRY0, "delete_all_entries"));
4881
4882         if (mp->start == NULL) {
4883                 bam_print(EMPTY_MENU);
4884                 return (BAM_SUCCESS);
4885         }
4886
4887         if (delete_boot_entry(mp, ALL_ENTRIES, DBE_PRINTERR) != BAM_SUCCESS) {
4888                 return (BAM_ERROR);
4889         }
4890
4891         return (BAM_WRITE);
4892 }
4893
4894 static FILE *
4895 create_diskmap(char *osroot)
4896 {
4897         FILE *fp;
4898         char cmd[PATH_MAX + 16];
4899         char path[PATH_MAX];
4900         const char *fcn = "create_diskmap()";
4901
4902         /* make sure we have a map file */
4903         fp = fopen(GRUBDISK_MAP, "r");
4904         if (fp == NULL) {
4905                 int     ret;
4906
4907                 ret = snprintf(path, sizeof (path), "%s/%s", osroot,
4908                     CREATE_DISKMAP);
4909                 if (ret >= sizeof (path)) {
4910                         bam_error(PATH_TOO_LONG, osroot);
4911                         return (NULL);
4912                 }
4913                 if (is_safe_exec(path) == BAM_ERROR)
4914                         return (NULL);
4915
4916                 (void) snprintf(cmd, sizeof (cmd),
4917                     "%s/%s > /dev/null", osroot, CREATE_DISKMAP);
4918                 if (exec_cmd(cmd, NULL) != 0)
4919                         return (NULL);
4920                 fp = fopen(GRUBDISK_MAP, "r");
4921                 INJECT_ERROR1("DISKMAP_CREATE_FAIL", fp = NULL);
4922                 if (fp) {
4923                         BAM_DPRINTF((D_CREATED_DISKMAP, fcn, GRUBDISK_MAP));
```



```
4924                 } else {
4925                         BAM_DPRINTF((D_CREATE_DISKMAP_FAIL, fcn, GRUBDISK_MAP));
4926                 }
4927         }
4928         return (fp);
4929 }
4930
4931 #define SECTOR_SIZE     512
4932
4933 static int
4934 get_partition(char *device)
4935 {
4936         int i, fd, is_pcfs, partno = -1;
4937         struct mboot *mboot;
4938         char boot_sect[SECTOR_SIZE];
4939         char *wholedisk, *slice;
4940 #ifdef i386
4941         ext_part_t *epp;
4942         uint32_t secnum, numsec;
4943         int rval, pno, ext_partno = -1;
4944 #endif
4945
4946         /* form whole disk (p0) */
4947         slice = device + strlen(device) - 2;
4948         is_pcfs = (*slice != 's');
4949         if (!is_pcfs)
4950                 *slice = '\0';
4951         wholedisk = s_calloc(1, strlen(device) + 3);
4952         (void) snprintf(wholedisk, strlen(device) + 3, "%sp0", device);
4953         if (!is_pcfs)
4954                 *slice = 's';
4955
4956         /* read boot sector */
4957         fd = open(wholedisk, O_RDONLY);
4958         if (fd == -1 || read(fd, boot_sect, SECTOR_SIZE) != SECTOR_SIZE) {
4959                 return (partno);
4960         }
4961         (void) close(fd);
4962
4963 #ifdef i386
4964         /* Read/Initialize extended partition information */
4965         if ((rval = libfdisk_init(&epp, wholedisk, NULL, FDISK_READ_DISK))
4966             != FDISK_SUCCESS) {
4967                 switch (rval) {
4968                         /*
4969                          * FDISK_EBADLOGDRIVE and FDISK_ENOLOGDRIVE can
4970                          * be considered as soft errors and hence
4971                          * we do not return
4972                          */
4973                         case FDISK_EBADLOGDRIVE:
4974                                 break;
4975                         case FDISK_ENOLOGDRIVE:
4976                                 break;
4977                         case FDISK_EBADMAGIC:
4978                                 /*FALLTHROUGH*/
4979                         default:
4980                                 free(wholedisk);
4981                                 libfdisk_fini(&epp);
4982                                 return (partno);
4983                 }
4984         }
4985 #endif
4986         free(wholedisk);
4987
4988         /* parse fdisk table */
4989         mboot = (struct mboot *)((void *)boot_sect);
```

```
4990         for (i = 0; i < FD_NUMPART; i++) {
4991                 struct ipart *part =
4992                     (struct ipart *)(uintptr_t)mboot->parts + i;
4993                 if (is_pcfs) {  /* looking for solaris boot part */
4994                         if (part->systid == 0xbe) {
4995                                 partno = i;
4996                                 break;
4997                         }
4998                 } else {        /* look for solaris partition, old and new */
4999 #ifdef i386
5000                         if ((part->systid == SUNIXOS &&
5001                             (fdisk_is_linux_swap(epp, part->relsect,
5002                             NULL) != 0)) || part->systid == SUNIXOS2) {
5003 #else
5004                         if (part->systid == SUNIXOS ||
5005                             part->systid == SUNIXOS2) {
5006 #endif
5007                                 partno = i;
5008                                 break;
5009                         }

5011 #ifdef i386
5012                         if (fdisk_is_dos_extended(part->systid))
5013                                 ext_partno = i;
5014 #endif
5015                 }
5016         }
5017 #ifdef i386
5018         /* If no primary solaris partition, check extended partition */
5019         if ((partno == -1) && (ext_partno != -1)) {
5020                 rval = fdisk_get_solaris_part(epp, &pno, &secnum, &numsec);
5021                 if (rval == FDISK_SUCCESS) {
5022                         partno = pno - 1;
5023                 }
5024         }
5025         libfdisk_fini(&epp);
5026 #endif
5027         return (partno);
5028 }

5030 char *
5031 get_grubroot(char *osroot, char *osdev, char *menu_root)
5032 {
5033         char            *grubroot;      /* (hd#,#,#) */
5034         char            *slice;
5035         char            *grubhd;
5036         int             fdiskpart;
5037         int             found = 0;
5038         char            *devname;
5039         char            *ctdname = strstr(osdev, "dsk/");
5040         char            linebuf[PATH_MAX];
5041         FILE            *fp;

5043         INJECT_ERROR1("GRUBROOT_INVALID_OSDEV", ctdname = NULL);
5044         if (ctdname == NULL) {
5045                 bam_error(INVALID_DEV_DSK, osdev);
5046                 return (NULL);
5047         }

5049         if (menu_root && !menu_on_bootdisk(osroot, menu_root)) {
5050                 /* menu bears no resemblance to our reality */
5051                 bam_error(CANNOT_GRUBROOT_BOOTDISK, osdev);
5052                 return (NULL);
5053         }

5055         ctdname += strlen("dsk/");
```



```
5056         slice = strrchr(ctdname, 's');
5057         if (slice)
5058                 *slice = '\0';

5060         fp = create_diskmap(osroot);
5061         if (fp == NULL) {
5062                 bam_error(DISKMAP_FAIL, osroot);
5063                 return (NULL);
5064         }

5066         rewind(fp);
5067         while (s_fgets(linebuf, sizeof (linebuf), fp) != NULL) {
5068                 grubhd = strtok(linebuf, " \t\n");
5069                 if (grubhd)
5070                         devname = strtok(NULL, " \t\n");
5071                 else
5072                         devname = NULL;
5073                 if (devname && strcmp(devname, ctdname) == 0) {
5074                         found = 1;
5075                         break;
5076                 }
5077         }

5079         if (slice)
5080                 *slice = 's';

5082         (void) fclose(fp);
5083         fp = NULL;

5085         INJECT_ERROR1("GRUBROOT_BIOSDEV_FAIL", found = 0);
5086         if (found == 0) {
5087                 bam_error(BIOSDEV_SKIP, osdev);
5088                 return (NULL);
5089         }

5091         fdiskpart = get_partition(osdev);
5092         INJECT_ERROR1("GRUBROOT_FDISK_FAIL", fdiskpart = -1);
5093         if (fdiskpart == -1) {
5094                 bam_error(FDISKPART_FAIL, osdev);
5095                 return (NULL);
5096         }

5098         grubroot = s_calloc(1, 10);
5099         if (slice) {
5100                 (void) snprintf(grubroot, 10, "(hd%s,%d,%c)",
5101                     grubhd, fdiskpart, slice[1] + 'a' - '0');
5102         } else
5103                 (void) snprintf(grubroot, 10, "(hd%s,%d)",
5104                     grubhd, fdiskpart);

5106         assert(fp == NULL);
5107         assert(strncmp(grubroot, "(hd", strlen("(hd")) == 0);
5108         return (grubroot);
5109 }

5111 static char *
5112 find_primary_common(char *mntpt, char *fstype)
5113 {
5114         char            signdir[PATH_MAX];
5115         char            tmpsign[MAXNAMELEN + 1];
5116         char            *lu;
5117         char            *ufs;
5118         char            *zfs;
5119         DIR             *dirp = NULL;
5120         struct dirent   *entp;
5121         struct stat     sb;
```

```
5122         const char      *fcn = "find_primary_common()";
5123
5124         (void) snprintf(signdir, sizeof (signdir), "%s/%s",
5125             mntpt, GRUBSIGN_DIR);
5126
5127         if (stat(signdir, &sb) == -1) {
5128                 BAM_DPRINTF((D_NO_SIGNDIR, fcn, signdir));
5129                 return (NULL);
5130         }
5131
5132         dirp = opendir(signdir);
5133         INJECT_ERROR1("SIGNDIR_OPENDIR_FAIL", dirp = NULL);
5134         if (dirp == NULL) {
5135                 bam_error(OPENDIR_FAILED, signdir, strerror(errno));
5136                 return (NULL);
5137         }
5138
5139         ufs = zfs = lu = NULL;
5140
5141         while (entp = readdir(dirp)) {
5142                 if (strcmp(entp->d_name, ".") == 0 ||
5143                     strcmp(entp->d_name, "..") == 0)
5144                         continue;
5145
5146                 (void) snprintf(tmpsign, sizeof (tmpsign), "%s", entp->d_name);
5147
5148                 if (lu == NULL &&
5149                     strncmp(tmpsign, GRUBSIGN_LU_PREFIX,
5150                     strlen(GRUBSIGN_LU_PREFIX)) == 0) {
5151                         lu = s_strdup(tmpsign);
5152                 }
5153
5154                 if (ufs == NULL &&
5155                     strncmp(tmpsign, GRUBSIGN_UFS_PREFIX,
5156                     strlen(GRUBSIGN_UFS_PREFIX)) == 0) {
5157                         ufs = s_strdup(tmpsign);
5158                 }
5159
5160                 if (zfs == NULL &&
5161                     strncmp(tmpsign, GRUBSIGN_ZFS_PREFIX,
5162                     strlen(GRUBSIGN_ZFS_PREFIX)) == 0) {
5163                         zfs = s_strdup(tmpsign);
5164                 }
5165         }
5166
5167         BAM_DPRINTF((D_EXIST_PRIMARY_SIGNS, fcn,
5168             zfs ? zfs : "NULL",
5169             ufs ? ufs : "NULL",
5170             lu ? lu : "NULL"));
5171
5172         if (dirp) {
5173                 (void) closedir(dirp);
5174                 dirp = NULL;
5175         }
5176
5177         if (strcmp(fstype, "ufs") == 0 && zfs) {
5178                 bam_error(SIGN_FSTYPE_MISMATCH, zfs, "ufs");
5179                 free(zfs);
5180                 zfs = NULL;
5181         } else if (strcmp(fstype, "zfs") == 0 && ufs) {
5182                 bam_error(SIGN_FSTYPE_MISMATCH, ufs, "zfs");
5183                 free(ufs);
5184                 ufs = NULL;
5185         }
5186
5187         assert(dirp == NULL);
```



```
5189         /* For now, we let Live Upgrade take care of its signature itself */
5190         if (lu) {
5191                 BAM_DPRINTF((D_FREEING_LU_SIGNS, fcn, lu));
5192                 free(lu);
5193                 lu = NULL;
5194         }
5195
5196         return (zfs ? zfs : ufs);
5197 }
5198
5199 static char *
5200 find_backup_common(char *mntpt, char *fstype)
5201 {
5202         FILE            *bfp = NULL;
5203         char            tmpsign[MAXNAMELEN + 1];
5204         char            backup[PATH_MAX];
5205         char            *ufs;
5206         char            *zfs;
5207         char            *lu;
5208         int             error;
5209         const char      *fcn = "find_backup_common()";
5210
5211         /*
5212          * We didn't find it in the primary directory.
5213          * Look at the backup
5214          */
5215         (void) snprintf(backup, sizeof (backup), "%s%s",
5216             mntpt, GRUBSIGN_BACKUP);
5217
5218         bfp = fopen(backup, "r");
5219         if (bfp == NULL) {
5220                 error = errno;
5221                 if (bam_verbose) {
5222                         bam_error(OPEN_FAIL, backup, strerror(error));
5223                 }
5224                 BAM_DPRINTF((D_OPEN_FAIL, fcn, backup, strerror(error)));
5225                 return (NULL);
5226         }
5227
5228         ufs = zfs = lu = NULL;
5229
5230         while (s_fgets(tmpsign, sizeof (tmpsign), bfp) != NULL) {
5231
5232                 if (lu == NULL &&
5233                     strncmp(tmpsign, GRUBSIGN_LU_PREFIX,
5234                     strlen(GRUBSIGN_LU_PREFIX)) == 0) {
5235                         lu = s_strdup(tmpsign);
5236                 }
5237
5238                 if (ufs == NULL &&
5239                     strncmp(tmpsign, GRUBSIGN_UFS_PREFIX,
5240                     strlen(GRUBSIGN_UFS_PREFIX)) == 0) {
5241                         ufs = s_strdup(tmpsign);
5242                 }
5243
5244                 if (zfs == NULL &&
5245                     strncmp(tmpsign, GRUBSIGN_ZFS_PREFIX,
5246                     strlen(GRUBSIGN_ZFS_PREFIX)) == 0) {
5247                         zfs = s_strdup(tmpsign);
5248                 }
5249         }
5250
5251         BAM_DPRINTF((D_EXIST_BACKUP_SIGNS, fcn,
5252             zfs ? zfs : "NULL",
5253             ufs ? ufs : "NULL",
```

```
5254             lu ? lu : "NULL"));

5256         if (bfp) {
5257                 (void) fclose(bfp);
5258                 bfp = NULL;
5259         }

5261         if (strcmp(fstype, "ufs") == 0 && zfs) {
5262                 bam_error(SIGN_FSTYPE_MISMATCH, zfs, "ufs");
5263                 free(zfs);
5264                 zfs = NULL;
5265         } else if (strcmp(fstype, "zfs") == 0 && ufs) {
5266                 bam_error(SIGN_FSTYPE_MISMATCH, ufs, "zfs");
5267                 free(ufs);
5268                 ufs = NULL;
5269         }

5271         assert(bfp == NULL);

5273         /* For now, we let Live Upgrade take care of its signature itself */
5274         if (lu) {
5275                 BAM_DPRINTF((D_FREEING_LU_SIGNS, fcn, lu));
5276                 free(lu);
5277                 lu = NULL;
5278         }

5280         return (zfs ? zfs : ufs);
5281 }

5283 static char *
5284 find_ufs_existing(char *osroot)
5285 {
5286         char            *sign;
5287         const char      *fcn = "find_ufs_existing()";

5289         sign = find_primary_common(osroot, "ufs");
5290         if (sign == NULL) {
5291                 sign = find_backup_common(osroot, "ufs");
5292                 BAM_DPRINTF((D_EXIST_BACKUP_SIGN, fcn, sign ? sign : "NULL"));
5293         } else {
5294                 BAM_DPRINTF((D_EXIST_PRIMARY_SIGN, fcn, sign));
5295         }

5297         return (sign);
5298 }

5300 char *
5301 get_mountpoint(char *special, char *fstype)
5302 {
5303         FILE            *mntfp;
5304         struct mnttab   mp = {0};
5305         struct mnttab   mpref = {0};
5306         int             error;
5307         int             ret;
5308         const char      *fcn = "get_mountpoint()";

5310         BAM_DPRINTF((D_FUNC_ENTRY2, fcn, special, fstype));

5312         mntfp = fopen(MNTTAB, "r");
5313         error = errno;
5314         INJECT_ERROR1("MNTTAB_ERR_GET_MNTPT", mntfp = NULL);
5315         if (mntfp == NULL) {
5316                 bam_error(OPEN_FAIL, MNTTAB, strerror(error));
5317                 return (NULL);
5318         }
```



```
5320         mpref.mnt_special = special;
5321         mpref.mnt_fstype = fstype;

5323         ret = getmntany(mntfp, &mp, &mpref);
5324         INJECT_ERROR1("GET_MOUNTPOINT_MNTANY", ret = 1);
5325         if (ret != 0) {
5326                 (void) fclose(mntfp);
5327                 BAM_DPRINTF((D_NO_MNTPT, fcn, special, fstype));
5328                 return (NULL);
5329         }
5330         (void) fclose(mntfp);

5332         assert(mp.mnt_mountp);

5334         BAM_DPRINTF((D_GET_MOUNTPOINT_RET, fcn, special, mp.mnt_mountp));

5336         return (s_strdup(mp.mnt_mountp));
5337 }

5339 /*
5340  * Mounts a "legacy" top dataset (if needed)
5341  * Returns:     The mountpoint of the legacy top dataset or NULL on error
5342  *              mnted returns one of the above values defined for zfs_mnted_t
5343  */
5344 static char *
5345 mount_legacy_dataset(char *pool, zfs_mnted_t *mnted)
5346 {
5347         char            cmd[PATH_MAX];
5348         char            tmpmnt[PATH_MAX];
5349         filelist_t      flist = {0};
5350         char            *is_mounted;
5351         struct stat     sb;
5352         int             ret;
5353         const char      *fcn = "mount_legacy_dataset()";

5355         BAM_DPRINTF((D_FUNC_ENTRY1, fcn, pool));

5357         *mnted = ZFS_MNT_ERROR;

5359         (void) snprintf(cmd, sizeof (cmd),
5360             "/sbin/zfs get -Ho value mounted %s",
5361             pool);

5363         ret = exec_cmd(cmd, &flist);
5364         INJECT_ERROR1("Z_MOUNT_LEG_GET_MOUNTED_CMD", ret = 1);
5365         if (ret != 0) {
5366                 bam_error(ZFS_MNTED_FAILED, pool);
5367                 return (NULL);
5368         }

5370         INJECT_ERROR1("Z_MOUNT_LEG_GET_MOUNTED_OUT", flist.head = NULL);
5371         if ((flist.head == NULL) || (flist.head != flist.tail)) {
5372                 bam_error(BAD_ZFS_MNTED, pool);
5373                 filelist_free(&flist);
5374                 return (NULL);
5375         }

5377         is_mounted = strtok(flist.head->line, " \t\n");
5378         INJECT_ERROR1("Z_MOUNT_LEG_GET_MOUNTED_STRTOK_YES", is_mounted = "yes");
5379         INJECT_ERROR1("Z_MOUNT_LEG_GET_MOUNTED_STRTOK_NO", is_mounted = "no");
5380         if (strcmp(is_mounted, "no") != 0) {
5381                 filelist_free(&flist);
5382                 *mnted = LEGACY_ALREADY;
5383                 /* get_mountpoint returns a strdup'ed string */
5384                 BAM_DPRINTF((D_Z_MOUNT_TOP_LEG_ALREADY, fcn, pool));
5385                 return (get_mountpoint(pool, "zfs"));
```

```
5386         }

5388         filelist_free(&flist);

5390         /*
5391          * legacy top dataset is not mounted. Mount it now
5392          * First create a mountpoint.
5393          */
5394         (void) snprintf(tmpmnt, sizeof (tmpmnt), "%s.%d",
5395             ZFS_LEGACY_MNTPT, getpid());

5397         ret = stat(tmpmnt, &sb);
5398         if (ret == -1) {
5399                 BAM_DPRINTF((D_Z_MOUNT_TOP_LEG_MNTPT_ABS, fcn, pool, tmpmnt));
5400                 ret = mkdirp(tmpmnt, DIR_PERMS);
5401                 INJECT_ERROR1("Z_MOUNT_TOP_LEG_MNTPT_MKDIRP", ret = -1);
5402                 if (ret == -1) {
5403                         bam_error(MKDIR_FAILED, tmpmnt, strerror(errno));
5404                         return (NULL);
5405                 }
5406         } else {
5407                 BAM_DPRINTF((D_Z_MOUNT_TOP_LEG_MNTPT_PRES, fcn, pool, tmpmnt));
5408         }

5410         (void) snprintf(cmd, sizeof (cmd),
5411             "/sbin/mount -F zfs %s %s",
5412             pool, tmpmnt);

5414         ret = exec_cmd(cmd, NULL);
5415         INJECT_ERROR1("Z_MOUNT_TOP_LEG_MOUNT_CMD", ret = 1);
5416         if (ret != 0) {
5417                 bam_error(ZFS_MOUNT_FAILED, pool);
5418                 (void) rmdir(tmpmnt);
5419                 return (NULL);
5420         }

5422         *mnted = LEGACY_MOUNTED;
5423         BAM_DPRINTF((D_Z_MOUNT_TOP_LEG_MOUNTED, fcn, pool, tmpmnt));
5424         return (s_strdup(tmpmnt));
5425 }

5427 /*
5428  * Mounts the top dataset (if needed)
5429  * Returns:     The mountpoint of the top dataset or NULL on error
5430  *              mnted returns one of the above values defined for zfs_mnted_t
5431  */
5432 static char *
5433 mount_top_dataset(char *pool, zfs_mnted_t *mnted)
5434 {
5435         char            cmd[PATH_MAX];
5436         filelist_t      flist = {0};
5437         char            *is_mounted;
5438         char            *mntpt;
5439         char            *zmntpt;
5440         int             ret;
5441         const char      *fcn = "mount_top_dataset()";

5443         *mnted = ZFS_MNT_ERROR;

5445         BAM_DPRINTF((D_FUNC_ENTRY1, fcn, pool));

5447         /*
5448          * First check if the top dataset is a "legacy" dataset
5449          */
5450         (void) snprintf(cmd, sizeof (cmd),
5451             "/sbin/zfs get -Ho value mountpoint %s",
```



```
5452             pool);
5453         ret = exec_cmd(cmd, &flist);
5454         INJECT_ERROR1("Z_MOUNT_TOP_GET_MNTPT", ret = 1);
5455         if (ret != 0) {
5456                 bam_error(ZFS_MNTPT_FAILED, pool);
5457                 return (NULL);
5458         }

5460         if (flist.head && (flist.head == flist.tail)) {
5461                 char *legacy = strtok(flist.head->line, " \t\n");
5462                 if (legacy && strcmp(legacy, "legacy") == 0) {
5463                         filelist_free(&flist);
5464                         BAM_DPRINTF((D_Z_IS_LEGACY, fcn, pool));
5465                         return (mount_legacy_dataset(pool, mnted));
5466                 }
5467         }

5469         filelist_free(&flist);

5471         BAM_DPRINTF((D_Z_IS_NOT_LEGACY, fcn, pool));

5473         (void) snprintf(cmd, sizeof (cmd),
5474             "/sbin/zfs get -Ho value mounted %s",
5475             pool);

5477         ret = exec_cmd(cmd, &flist);
5478         INJECT_ERROR1("Z_MOUNT_TOP_NONLEG_GET_MOUNTED", ret = 1);
5479         if (ret != 0) {
5480                 bam_error(ZFS_MNTED_FAILED, pool);
5481                 return (NULL);
5482         }

5484         INJECT_ERROR1("Z_MOUNT_TOP_NONLEG_GET_MOUNTED_VAL", flist.head = NULL);
5485         if ((flist.head == NULL) || (flist.head != flist.tail)) {
5486                 bam_error(BAD_ZFS_MNTED, pool);
5487                 filelist_free(&flist);
5488                 return (NULL);
5489         }

5491         is_mounted = strtok(flist.head->line, " \t\n");
5492         INJECT_ERROR1("Z_MOUNT_TOP_NONLEG_GET_MOUNTED_YES", is_mounted = "yes");
5493         INJECT_ERROR1("Z_MOUNT_TOP_NONLEG_GET_MOUNTED_NO", is_mounted = "no");
5494         if (strcmp(is_mounted, "no") != 0) {
5495                 filelist_free(&flist);
5496                 *mnted = ZFS_ALREADY;
5497                 BAM_DPRINTF((D_Z_MOUNT_TOP_NONLEG_MOUNTED_ALREADY, fcn, pool));
5498                 goto mounted;
5499         }

5501         filelist_free(&flist);
5502         BAM_DPRINTF((D_Z_MOUNT_TOP_NONLEG_MOUNTED_NOT_ALREADY, fcn, pool));

5504         /* top dataset is not mounted. Mount it now */
5505         (void) snprintf(cmd, sizeof (cmd),
5506             "/sbin/zfs mount %s", pool);
5507         ret = exec_cmd(cmd, NULL);
5508         INJECT_ERROR1("Z_MOUNT_TOP_NONLEG_MOUNT_CMD", ret = 1);
5509         if (ret != 0) {
5510                 bam_error(ZFS_MOUNT_FAILED, pool);
5511                 return (NULL);
5512         }
5513         *mnted = ZFS_MOUNTED;
5514         BAM_DPRINTF((D_Z_MOUNT_TOP_NONLEG_MOUNTED_NOW, fcn, pool));
5515         /*FALLTHRU*/
5516 mounted:
5517         /*
```

```
5518          * Now get the mountpoint
5519          */
5520         (void) snprintf(cmd, sizeof (cmd),
5521             "/sbin/zfs get -Ho value mountpoint %s",
5522             pool);

5524         ret = exec_cmd(cmd, &flist);
5525         INJECT_ERROR1("Z_MOUNT_TOP_NONLEG_GET_MNTPT_CMD", ret = 1);
5526         if (ret != 0) {
5527                 bam_error(ZFS_MNTPT_FAILED, pool);
5528                 goto error;
5529         }

5531         INJECT_ERROR1("Z_MOUNT_TOP_NONLEG_GET_MNTPT_OUT", flist.head = NULL);
5532         if ((flist.head == NULL) || (flist.head != flist.tail)) {
5533                 bam_error(NULL_ZFS_MNTPT, pool);
5534                 goto error;
5535         }

5537         mntpt = strtok(flist.head->line, " \t\n");
5538         INJECT_ERROR1("Z_MOUNT_TOP_NONLEG_GET_MNTPT_STRTOK", mntpt = "foo");
5539         if (*mntpt != '/') {
5540                 bam_error(BAD_ZFS_MNTPT, pool, mntpt);
5541                 goto error;
5542         }
5543         zmntpt = s_strdup(mntpt);

5545         filelist_free(&flist);

5547         BAM_DPRINTF((D_Z_MOUNT_TOP_NONLEG_MNTPT, fcn, pool, zmntpt));

5549         return (zmntpt);

5551 error:
5552         filelist_free(&flist);
5553         (void) umount_top_dataset(pool, *mnted, NULL);
5554         BAM_DPRINTF((D_RETURN_FAILURE, fcn));
5555         return (NULL);
5556 }

5558 static int
5559 umount_top_dataset(char *pool, zfs_mnted_t mnted, char *mntpt)
5560 {
5561         char            cmd[PATH_MAX];
5562         int             ret;
5563         const char      *fcn = "umount_top_dataset()";

5565         INJECT_ERROR1("Z_UMOUNT_TOP_INVALID_STATE", mnted = ZFS_MNT_ERROR);
5566         switch (mnted) {
5567         case LEGACY_ALREADY:
5568         case ZFS_ALREADY:
5569                 /* nothing to do */
5570                 BAM_DPRINTF((D_Z_UMOUNT_TOP_ALREADY_NOP, fcn, pool,
5571                     mntpt ? mntpt : "NULL"));
5572                 free(mntpt);
5573                 return (BAM_SUCCESS);
5574         case LEGACY_MOUNTED:
5575                 (void) snprintf(cmd, sizeof (cmd),
5576                     "/sbin/umount %s", pool);
5577                 ret = exec_cmd(cmd, NULL);
5578                 INJECT_ERROR1("Z_UMOUNT_TOP_LEGACY_UMOUNT_FAIL", ret = 1);
5579                 if (ret != 0) {
5580                         bam_error(UMOUNT_FAILED, pool);
5581                         free(mntpt);
5582                         return (BAM_ERROR);
5583                 }
```



```
5584                 if (mntpt)
5585                         (void) rmdir(mntpt);
5586                 free(mntpt);
5587                 BAM_DPRINTF((D_Z_UMOUNT_TOP_LEGACY, fcn, pool));
5588                 return (BAM_SUCCESS);
5589         case ZFS_MOUNTED:
5590                 free(mntpt);
5591                 (void) snprintf(cmd, sizeof (cmd),
5592                     "/sbin/zfs unmount %s", pool);
5593                 ret = exec_cmd(cmd, NULL);
5594                 INJECT_ERROR1("Z_UMOUNT_TOP_NONLEG_UMOUNT_FAIL", ret = 1);
5595                 if (ret != 0) {
5596                         bam_error(UMOUNT_FAILED, pool);
5597                         return (BAM_ERROR);
5598                 }
5599                 BAM_DPRINTF((D_Z_UMOUNT_TOP_NONLEG, fcn, pool));
5600                 return (BAM_SUCCESS);
5601         default:
5602                 bam_error(INT_BAD_MNTSTATE, pool);
5603                 return (BAM_ERROR);
5604         }
5605         /*NOTREACHED*/
5606 }

5608 /*
5609  * For ZFS, osdev can be one of two forms
5610  * It can be a "special" file as seen in mnttab: rpool/ROOT/szboot_0402
5611  * It can be a /dev/[r]dsk special file. We handle both instances
5612  */
5613 static char *
5614 get_pool(char *osdev)
5615 {
5616         char            cmd[PATH_MAX];
5617         char            buf[PATH_MAX];
5618         filelist_t      flist = {0};
5619         char            *pool;
5620         char            *cp;
5621         char            *slash;
5622         int             ret;
5623         const char      *fcn = "get_pool()";

5625         INJECT_ERROR1("GET_POOL_OSDEV", osdev = NULL);
5626         if (osdev == NULL) {
5627                 bam_error(GET_POOL_OSDEV_NULL);
5628                 return (NULL);
5629         }

5631         BAM_DPRINTF((D_GET_POOL_OSDEV, fcn, osdev));

5633         if (osdev[0] != '/') {
5634                 (void) strlcpy(buf, osdev, sizeof (buf));
5635                 slash = strchr(buf, '/');
5636                 if (slash)
5637                         *slash = '\0';
5638                 pool = s_strdup(buf);
5639                 BAM_DPRINTF((D_GET_POOL_RET, fcn, pool));
5640                 return (pool);
5641         } else if (strncmp(osdev, "/dev/dsk/", strlen("/dev/dsk/")) != 0 &&
5642             strncmp(osdev, "/dev/rdsk/", strlen("/dev/rdsk/")) != 0) {
5643                 bam_error(GET_POOL_BAD_OSDEV, osdev);
5644                 return (NULL);
5645         }

5647         /*
5648          * Call the zfs fstyp directly since this is a zpool. This avoids
5649          * potential pcfs conflicts if the first block wasn't cleared.
```

```
5650          */
5651         (void) snprintf(cmd, sizeof (cmd),
5652             "/usr/lib/fs/zfs/fstyp -a %s 2>/dev/null | /bin/grep '^name:'",
5653             osdev);
5654
5655         ret = exec_cmd(cmd, &flist);
5656         INJECT_ERROR1("GET_POOL_FSTYP", ret = 1);
5657         if (ret != 0) {
5658                 bam_error(FSTYP_A_FAILED, osdev);
5659                 return (NULL);
5660         }
5661
5662         INJECT_ERROR1("GET_POOL_FSTYP_OUT", flist.head = NULL);
5663         if ((flist.head == NULL) || (flist.head != flist.tail)) {
5664                 bam_error(NULL_FSTYP_A, osdev);
5665                 filelist_free(&flist);
5666                 return (NULL);
5667         }
5668
5669         (void) strtok(flist.head->line, "'");
5670         cp = strtok(NULL, "'");
5671         INJECT_ERROR1("GET_POOL_FSTYP_STRTOK", cp = NULL);
5672         if (cp == NULL) {
5673                 bam_error(BAD_FSTYP_A, osdev);
5674                 filelist_free(&flist);
5675                 return (NULL);
5676         }
5677
5678         pool = s_strdup(cp);
5679
5680         filelist_free(&flist);
5681
5682         BAM_DPRINTF((D_GET_POOL_RET, fcn, pool));
5683
5684         return (pool);
5685 }
5686
5687 static char *
5688 find_zfs_existing(char *osdev)
5689 {
5690         char            *pool;
5691         zfs_mnted_t     mnted;
5692         char            *mntpt;
5693         char            *sign;
5694         const char      *fcn = "find_zfs_existing()";
5695
5696         pool = get_pool(osdev);
5697         INJECT_ERROR1("ZFS_FIND_EXIST_POOL", pool = NULL);
5698         if (pool == NULL) {
5699                 bam_error(ZFS_GET_POOL_FAILED, osdev);
5700                 return (NULL);
5701         }
5702
5703         mntpt = mount_top_dataset(pool, &mnted);
5704         INJECT_ERROR1("ZFS_FIND_EXIST_MOUNT_TOP", mntpt = NULL);
5705         if (mntpt == NULL) {
5706                 bam_error(ZFS_MOUNT_TOP_DATASET_FAILED, pool);
5707                 free(pool);
5708                 return (NULL);
5709         }
5710
5711         sign = find_primary_common(mntpt, "zfs");
5712         if (sign == NULL) {
5713                 sign = find_backup_common(mntpt, "zfs");
5714                 BAM_DPRINTF((D_EXIST_BACKUP_SIGN, fcn, sign ? sign : "NULL"));
5715         } else {
```



```
5716                 BAM_DPRINTF((D_EXIST_PRIMARY_SIGN, fcn, sign));
5717         }
5718
5719         (void) umount_top_dataset(pool, mnted, mntpt);
5720
5721         free(pool);
5722
5723         return (sign);
5724 }
5725
5726 static char *
5727 find_existing_sign(char *osroot, char *osdev, char *fstype)
5728 {
5729         const char              *fcn = "find_existing_sign()";
5730
5731         INJECT_ERROR1("FIND_EXIST_NOTSUP_FS", fstype = "foofs");
5732         if (strcmp(fstype, "ufs") == 0) {
5733                 BAM_DPRINTF((D_CHECK_UFS_EXIST_SIGN, fcn));
5734                 return (find_ufs_existing(osroot));
5735         } else if (strcmp(fstype, "zfs") == 0) {
5736                 BAM_DPRINTF((D_CHECK_ZFS_EXIST_SIGN, fcn));
5737                 return (find_zfs_existing(osdev));
5738         } else {
5739                 bam_error(GRUBSIGN_NOTSUP, fstype);
5740                 return (NULL);
5741         }
5742 }
5743
5744 #define MH_HASH_SZ      16
5745
5746 typedef enum {
5747         MH_ERROR = -1,
5748         MH_NOMATCH,
5749         MH_MATCH
5750 } mh_search_t;
5751
5752 typedef struct mcache {
5753         char    *mc_special;
5754         char    *mc_mntpt;
5755         char    *mc_fstype;
5756         struct mcache *mc_next;
5757 } mcache_t;
5758
5759 typedef struct mhash {
5760         mcache_t *mh_hash[MH_HASH_SZ];
5761 } mhash_t;
5762
5763 static int
5764 mhash_fcn(char *key)
5765 {
5766         int             i;
5767         uint64_t        sum = 0;
5768
5769         for (i = 0; key[i] != '\0'; i++) {
5770                 sum += (uchar_t)key[i];
5771         }
5772
5773         sum %= MH_HASH_SZ;
5774
5775         assert(sum < MH_HASH_SZ);
5776
5777         return (sum);
5778 }
5779
5780 static mhash_t *
5781 cache_mnttab(void)
```

```
5782 {
5783         FILE            *mfp;
5784         struct extmnttab mnt;
5785         mcache_t        *mcp;
5786         mhash_t         *mhp;
5787         char            *ctds;
5788         int             idx;
5789         int             error;
5790         char            *special_dup;
5791         const char      *fcn = "cache_mnttab()";

5793         mfp = fopen(MNTTAB, "r");
5794         error = errno;
5795         INJECT_ERROR1("CACHE_MNTTAB_MNTTAB_ERR", mfp = NULL);
5796         if (mfp == NULL) {
5797                 bam_error(OPEN_FAIL, MNTTAB, strerror(error));
5798                 return (NULL);
5799         }

5801         mhp = s_calloc(1, sizeof (mhash_t));

5803         resetmnttab(mfp);

5805         while (getextmntent(mfp, &mnt, sizeof (mnt)) == 0) {
5806                 /* only cache ufs */
5807                 if (strcmp(mnt.mnt_fstype, "ufs") != 0)
5808                         continue;

5810                 /* basename() modifies its arg, so dup it */
5811                 special_dup = s_strdup(mnt.mnt_special);
5812                 ctds = basename(special_dup);

5814                 mcp = s_calloc(1, sizeof (mcache_t));
5815                 mcp->mc_special = s_strdup(ctds);
5816                 mcp->mc_mntpt = s_strdup(mnt.mnt_mountp);
5817                 mcp->mc_fstype = s_strdup(mnt.mnt_fstype);
5818                 BAM_DPRINTF((D_CACHE_MNTS, fcn, ctds,
5819                     mnt.mnt_mountp, mnt.mnt_fstype));
5820                 idx = mhash_fcn(ctds);
5821                 mcp->mc_next = mhp->mh_hash[idx];
5822                 mhp->mh_hash[idx] = mcp;
5823                 free(special_dup);
5824         }

5826         (void) fclose(mfp);

5828         return (mhp);
5829 }

5831 static void
5832 free_mnttab(mhash_t *mhp)
5833 {
5834         mcache_t        *mcp;
5835         int             i;

5837         for (i = 0; i < MH_HASH_SZ; i++) {
5838                 /*LINTED*/
5839                 while (mcp = mhp->mh_hash[i]) {
5840                         mhp->mh_hash[i] = mcp->mc_next;
5841                         free(mcp->mc_special);
5842                         free(mcp->mc_mntpt);
5843                         free(mcp->mc_fstype);
5844                         free(mcp);
5845                 }
5846         }
```



```
5848         for (i = 0; i < MH_HASH_SZ; i++) {
5849                 assert(mhp->mh_hash[i] == NULL);
5850         }
5851         free(mhp);
5852 }

5854 static mh_search_t
5855 search_hash(mhash_t *mhp, char *special, char **mntpt)
5856 {
5857         int             idx;
5858         mcache_t        *mcp;
5859         const char      *fcn = "search_hash()";

5861         assert(mntpt);

5863         *mntpt = NULL;

5865         INJECT_ERROR1("SEARCH_HASH_FULL_PATH", special = "/foo");
5866         if (strchr(special, '/')) {
5867                 bam_error(INVALID_MHASH_KEY, special);
5868                 return (MH_ERROR);
5869         }

5871         idx = mhash_fcn(special);

5873         for (mcp = mhp->mh_hash[idx]; mcp; mcp = mcp->mc_next) {
5874                 if (strcmp(mcp->mc_special, special) == 0)
5875                         break;
5876         }

5878         if (mcp == NULL) {
5879                 BAM_DPRINTF((D_MNTTAB_HASH_NOMATCH, fcn, special));
5880                 return (MH_NOMATCH);
5881         }

5883         assert(strcmp(mcp->mc_fstype, "ufs") == 0);
5884         *mntpt = mcp->mc_mntpt;
5885         BAM_DPRINTF((D_MNTTAB_HASH_MATCH, fcn, special));
5886         return (MH_MATCH);
5887 }

5889 static int
5890 check_add_ufs_sign_to_list(FILE *tfp, char *mntpt)
5891 {
5892         char            *sign;
5893         char            *signline;
5894         char            signbuf[MAXNAMELEN];
5895         int             len;
5896         int             error;
5897         const char      *fcn = "check_add_ufs_sign_to_list()";

5899         /* safe to specify NULL as "osdev" arg for UFS */
5900         sign = find_existing_sign(mntpt, NULL, "ufs");
5901         if (sign == NULL) {
5902                 /* No existing signature, nothing to add to list */
5903                 BAM_DPRINTF((D_NO_SIGN_TO_LIST, fcn, mntpt));
5904                 return (0);
5905         }

5907         (void) snprintf(signbuf, sizeof (signbuf), "%s\n", sign);
5908         signline = signbuf;

5910         INJECT_ERROR1("UFS_MNTPT_SIGN_NOTUFS", signline = "pool_rpool10\n");
5911         if (strncmp(signline, GRUBSIGN_UFS_PREFIX,
5912             strlen(GRUBSIGN_UFS_PREFIX))) {
5913                 bam_error(INVALID_UFS_SIGNATURE, sign);
```

```
5914                free(sign);
5915                /* ignore invalid signatures */
5916                return (0);
5917        }

5919        len = fputs(signline, tfp);
5920        error = errno;
5921        INJECT_ERROR1("SIGN_LIST_PUTS_ERROR", len = 0);
5922        if (len != strlen(signline)) {
5923                bam_error(SIGN_LIST_FPUTS_ERR, sign, strerror(error));
5924                free(sign);
5925                return (-1);
5926        }

5928        free(sign);

5930        BAM_DPRINTF((D_SIGN_LIST_PUTS_DONE, fcn, mntpt));
5931        return (0);
5932 }

5934 /*
5935  * slice is a basename not a full pathname
5936  */
5937 static int
5938 process_slice_common(char *slice, FILE *tfp, mhash_t *mhp, char *tmpmnt)
5939 {
5940        int             ret;
5941        char            cmd[PATH_MAX];
5942        char            path[PATH_MAX];
5943        struct stat     sbuf;
5944        char            *mntpt;
5945        filelist_t      flist = {0};
5946        char            *fstype;
5947        char            blkslice[PATH_MAX];
5948        const char      *fcn = "process_slice_common()";


5951        ret = search_hash(mhp, slice, &mntpt);
5952        switch (ret) {
5953                case MH_MATCH:
5954                        if (check_add_ufs_sign_to_list(tfp, mntpt) == -1)
5955                                return (-1);
5956                        else
5957                                return (0);
5958                case MH_NOMATCH:
5959                        break;
5960                case MH_ERROR:
5961                default:
5962                        return (-1);
5963        }

5965        (void) snprintf(path, sizeof (path), "/dev/rdsk/%s", slice);
5966        if (stat(path, &sbuf) == -1) {
5967                BAM_DPRINTF((D_SLICE_ENOENT, fcn, path));
5968                return (0);
5969        }

5971        /* Check if ufs. Call ufs fstyp directly to avoid pcfs conflicts. */
5972        (void) snprintf(cmd, sizeof (cmd),
5973            "/usr/lib/fs/ufs/fstyp /dev/rdsk/%s 2>/dev/null",
5974            slice);

5976        if (exec_cmd(cmd, &flist) != 0) {
5977                if (bam_verbose)
5978                        bam_print(FSTYP_FAILED, slice);
5979                return (0);
```



```
5980        }

5982        if ((flist.head == NULL) || (flist.head != flist.tail)) {
5983                if (bam_verbose)
5984                        bam_print(FSTYP_BAD, slice);
5985                filelist_free(&flist);
5986                return (0);
5987        }

5989        fstype = strtok(flist.head->line, " \t\n");
5990        if (fstype == NULL || strcmp(fstype, "ufs") != 0) {
5991                if (bam_verbose)
5992                        bam_print(NOT_UFS_SLICE, slice, fstype);
5993                filelist_free(&flist);
5994                return (0);
5995        }

5997        filelist_free(&flist);

5999        /*
6000         * Since we are mounting the filesystem read-only, the
6001         * the last mount field of the superblock is unchanged
6002         * and does not need to be fixed up post-mount;
6003         */

6005        (void) snprintf(blkslice, sizeof (blkslice), "/dev/dsk/%s",
6006            slice);

6008        (void) snprintf(cmd, sizeof (cmd),
6009            "/usr/sbin/mount -F ufs -o ro %s %s "
6010            "> /dev/null 2>&1", blkslice, tmpmnt);

6012        if (exec_cmd(cmd, NULL) != 0) {
6013                if (bam_verbose)
6014                        bam_print(MOUNT_FAILED, blkslice, "ufs");
6015                return (0);
6016        }

6018        ret = check_add_ufs_sign_to_list(tfp, tmpmnt);

6020        (void) snprintf(cmd, sizeof (cmd),
6021            "/usr/sbin/umount -f %s > /dev/null 2>&1",
6022            tmpmnt);

6024        if (exec_cmd(cmd, NULL) != 0) {
6025                bam_print(UMOUNT_FAILED, slice);
6026                return (0);
6027        }

6029        return (ret);
6030 }

6032 static int
6033 process_vtoc_slices(
6034        char *s0,
6035        struct vtoc *vtoc,
6036        FILE *tfp,
6037        mhash_t *mhp,
6038        char *tmpmnt)
6039 {
6040        int             idx;
6041        char            slice[PATH_MAX];
6042        size_t          len;
6043        char            *cp;
6044        const char      *fcn = "process_vtoc_slices()";
```

```
6046         len = strlen(s0);
6048         assert(s0[len - 2] == 's' && s0[len - 1] == '0');
6050         s0[len - 1] = '\0';
6052         (void) strlcpy(slice, s0, sizeof (slice));
6054         s0[len - 1] = '0';
6056         cp = slice + len - 1;
6058         for (idx = 0; idx < vtoc->v_nparts; idx++) {
6060                 (void) snprintf(cp, sizeof (slice) - (len - 1), "%u", idx);
6062                 if (vtoc->v_part[idx].p_size == 0) {
6063                         BAM_DPRINTF((D_VTOC_SIZE_ZERO, fcn, slice));
6064                         continue;
6065                 }
6067                 /* Skip "SWAP", "USR", "BACKUP", "VAR", "HOME", "ALTSCTR" */
6068                 switch (vtoc->v_part[idx].p_tag) {
6069                 case V_SWAP:
6070                 case V_USR:
6071                 case V_BACKUP:
6072                 case V_VAR:
6073                 case V_HOME:
6074                 case V_ALTSCTR:
6075                         BAM_DPRINTF((D_VTOC_NOT_ROOT_TAG, fcn, slice));
6076                         continue;
6077                 default:
6078                         BAM_DPRINTF((D_VTOC_ROOT_TAG, fcn, slice));
6079                         break;
6080                 }
6082                 /* skip unmountable and readonly slices */
6083                 switch (vtoc->v_part[idx].p_flag) {
6084                 case V_UNMNT:
6085                 case V_RONLY:
6086                         BAM_DPRINTF((D_VTOC_NOT_RDWR_FLAG, fcn, slice));
6087                         continue;
6088                 default:
6089                         BAM_DPRINTF((D_VTOC_RDWR_FLAG, fcn, slice));
6090                         break;
6091                 }
6093                 if (process_slice_common(slice, tfp, mhp, tmpmnt) == -1) {
6094                         return (-1);
6095                 }
6096         }
6098         return (0);
6099 }
6101 static int
6102 process_efi_slices(
6103         char *s0,
6104         struct dk_gpt *efi,
6105         FILE *tfp,
6106         mhash_t *mhp,
6107         char *tmpmnt)
6108 {
6109         int             idx;
6110         char            slice[PATH_MAX];
6111         size_t          len;
```



```
6112         char            *cp;
6113         const char      *fcn = "process_efi_slices()";
6115         len = strlen(s0);
6117         assert(s0[len - 2] == 's' && s0[len - 1] == '0');
6119         s0[len - 1] = '\0';
6121         (void) strlcpy(slice, s0, sizeof (slice));
6123         s0[len - 1] = '0';
6125         cp = slice + len - 1;
6127         for (idx = 0; idx < efi->efi_nparts; idx++) {
6129                 (void) snprintf(cp, sizeof (slice) - (len - 1), "%u", idx);
6131                 if (efi->efi_parts[idx].p_size == 0) {
6132                         BAM_DPRINTF((D_EFI_SIZE_ZERO, fcn, slice));
6133                         continue;
6134                 }
6136                 /* Skip "SWAP", "USR", "BACKUP", "VAR", "HOME", "ALTSCTR" */
6137                 switch (efi->efi_parts[idx].p_tag) {
6138                 case V_SWAP:
6139                 case V_USR:
6140                 case V_BACKUP:
6141                 case V_VAR:
6142                 case V_HOME:
6143                 case V_ALTSCTR:
6144                         BAM_DPRINTF((D_EFI_NOT_ROOT_TAG, fcn, slice));
6145                         continue;
6146                 default:
6147                         BAM_DPRINTF((D_EFI_ROOT_TAG, fcn, slice));
6148                         break;
6149                 }
6151                 /* skip unmountable and readonly slices */
6152                 switch (efi->efi_parts[idx].p_flag) {
6153                 case V_UNMNT:
6154                 case V_RONLY:
6155                         BAM_DPRINTF((D_EFI_NOT_RDWR_FLAG, fcn, slice));
6156                         continue;
6157                 default:
6158                         BAM_DPRINTF((D_EFI_RDWR_FLAG, fcn, slice));
6159                         break;
6160                 }
6162                 if (process_slice_common(slice, tfp, mhp, tmpmnt) == -1) {
6163                         return (-1);
6164                 }
6165         }
6167         return (0);
6168 }
6170 /*
6171  * s0 is a basename not a full path
6172  */
6173 static int
6174 process_slice0(char *s0, FILE *tfp, mhash_t *mhp, char *tmpmnt)
6175 {
6176         struct vtoc             vtoc;
6177         struct dk_gpt           *efi;
```

```
6178         char                    s0path[PATH_MAX];
6179         struct stat             sbuf;
6180         int                     e_flag;
6181         int                     v_flag;
6182         int                     retval;
6183         int                     err;
6184         int                     fd;
6185         const char              *fcn = "process_slice0()";

6187         (void) snprintf(s0path, sizeof (s0path), "/dev/rdsk/%s", s0);

6189         if (stat(s0path, &sbuf) == -1) {
6190                 BAM_DPRINTF((D_SLICE0_ENOENT, fcn, s0path));
6191                 return (0);
6192         }

6194         fd = open(s0path, O_NONBLOCK|O_RDONLY);
6195         if (fd == -1) {
6196                 bam_error(OPEN_FAIL, s0path, strerror(errno));
6197                 return (0);
6198         }

6200         e_flag = v_flag = 0;
6201         retval = ((err = read_vtoc(fd, &vtoc)) >= 0) ? 0 : err;
6202         switch (retval) {
6203                 case VT_EIO:
6204                         BAM_DPRINTF((D_VTOC_READ_FAIL, fcn, s0path));
6205                         break;
6206                 case VT_EINVAL:
6207                         BAM_DPRINTF((D_VTOC_INVALID, fcn, s0path));
6208                         break;
6209                 case VT_ERROR:
6210                         BAM_DPRINTF((D_VTOC_UNKNOWN_ERR, fcn, s0path));
6211                         break;
6212                 case VT_ENOTSUP:
6213                         e_flag = 1;
6214                         BAM_DPRINTF((D_VTOC_NOTSUP, fcn, s0path));
6215                         break;
6216                 case 0:
6217                         v_flag = 1;
6218                         BAM_DPRINTF((D_VTOC_READ_SUCCESS, fcn, s0path));
6219                         break;
6220                 default:
6221                         BAM_DPRINTF((D_VTOC_UNKNOWN_RETCODE, fcn, s0path));
6222                         break;
6223         }


6226         if (e_flag) {
6227                 e_flag = 0;
6228                 retval = ((err = efi_alloc_and_read(fd, &efi)) >= 0) ? 0 : err;
6229                 switch (retval) {
6230                 case VT_EIO:
6231                         BAM_DPRINTF((D_EFI_READ_FAIL, fcn, s0path));
6232                         break;
6233                 case VT_EINVAL:
6234                         BAM_DPRINTF((D_EFI_INVALID, fcn, s0path));
6235                         break;
6236                 case VT_ERROR:
6237                         BAM_DPRINTF((D_EFI_UNKNOWN_ERR, fcn, s0path));
6238                         break;
6239                 case VT_ENOTSUP:
6240                         BAM_DPRINTF((D_EFI_NOTSUP, fcn, s0path));
6241                         break;
6242                 case 0:
6243                         e_flag = 1;
```



```
6244                         BAM_DPRINTF((D_EFI_READ_SUCCESS, fcn, s0path));
6245                         break;
6246                 default:
6247                         BAM_DPRINTF((D_EFI_UNKNOWN_RETCODE, fcn, s0path));
6248                         break;
6249                 }
6250         }

6252         (void) close(fd);

6254         if (v_flag) {
6255                 retval = process_vtoc_slices(s0,
6256                     &vtoc, tfp, mhp, tmpmnt);
6257         } else if (e_flag) {
6258                 retval = process_efi_slices(s0,
6259                     efi, tfp, mhp, tmpmnt);
6260         } else {
6261                 BAM_DPRINTF((D_NOT_VTOC_OR_EFI, fcn, s0path));
6262                 return (0);
6263         }

6265         return (retval);
6266 }

6268 /*
6269  * Find and create a list of all existing UFS boot signatures
6270  */
6271 static int
6272 FindAllUfsSignatures(void)
6273 {
6274         mhash_t         *mnttab_hash;
6275         DIR             *dirp = NULL;
6276         struct dirent   *dp;
6277         char            tmpmnt[PATH_MAX];
6278         char            cmd[PATH_MAX];
6279         struct stat     sb;
6280         int             fd;
6281         FILE            *tfp;
6282         size_t          len;
6283         int             ret;
6284         int             error;
6285         const char      *fcn = "FindAllUfsSignatures()";

6287         if (stat(UFS_SIGNATURE_LIST, &sb) != -1)  {
6288                 bam_print(SIGNATURE_LIST_EXISTS, UFS_SIGNATURE_LIST);
6289                 return (0);
6290         }

6292         fd = open(UFS_SIGNATURE_LIST".tmp",
6293             O_RDWR|O_CREAT|O_TRUNC, 0644);
6294         error = errno;
6295         INJECT_ERROR1("SIGN_LIST_TMP_TRUNC", fd = -1);
6296         if (fd == -1) {
6297                 bam_error(OPEN_FAIL, UFS_SIGNATURE_LIST".tmp", strerror(error));
6298                 return (-1);
6299         }

6301         ret = close(fd);
6302         error = errno;
6303         INJECT_ERROR1("SIGN_LIST_TMP_CLOSE", ret = -1);
6304         if (ret == -1) {
6305                 bam_error(CLOSE_FAIL, UFS_SIGNATURE_LIST".tmp",
6306                     strerror(error));
6307                 (void) unlink(UFS_SIGNATURE_LIST".tmp");
6308                 return (-1);
6309         }
```

```
6311         tfp = fopen(UFS_SIGNATURE_LIST".tmp", "a");
6312         error = errno;
6313         INJECT_ERROR1("SIGN_LIST_APPEND_FOPEN", tfp = NULL);
6314         if (tfp == NULL) {
6315                 bam_error(OPEN_FAIL, UFS_SIGNATURE_LIST".tmp", strerror(error));
6316                 (void) unlink(UFS_SIGNATURE_LIST".tmp");
6317                 return (-1);
6318         }

6320         mnttab_hash = cache_mnttab();
6321         INJECT_ERROR1("CACHE_MNTTAB_ERROR", mnttab_hash = NULL);
6322         if (mnttab_hash == NULL) {
6323                 (void) fclose(tfp);
6324                 (void) unlink(UFS_SIGNATURE_LIST".tmp");
6325                 bam_error(CACHE_MNTTAB_FAIL, fcn);
6326                 return (-1);
6327         }

6329         (void) snprintf(tmpmnt, sizeof (tmpmnt),
6330             "/tmp/bootadm_ufs_sign_mnt.%d", getpid());
6331         (void) unlink(tmpmnt);

6333         ret = mkdirp(tmpmnt, DIR_PERMS);
6334         error = errno;
6335         INJECT_ERROR1("MKDIRP_SIGN_MNT", ret = -1);
6336         if (ret == -1) {
6337                 bam_error(MKDIR_FAILED, tmpmnt, strerror(error));
6338                 free_mnttab(mnttab_hash);
6339                 (void) fclose(tfp);
6340                 (void) unlink(UFS_SIGNATURE_LIST".tmp");
6341                 return (-1);
6342         }

6344         dirp = opendir("/dev/rdsk");
6345         error = errno;
6346         INJECT_ERROR1("OPENDIR_DEV_RDSK", dirp = NULL);
6347         if (dirp == NULL) {
6348                 bam_error(OPENDIR_FAILED, "/dev/rdsk", strerror(error));
6349                 goto fail;
6350         }

6352         while (dp = readdir(dirp)) {
6353                 if (strcmp(dp->d_name, ".") == 0 ||
6354                     strcmp(dp->d_name, "..") == 0)
6355                         continue;

6357                 /*
6358                  * we only look for the s0 slice. This is guranteed to
6359                  * have 's' at len - 2.
6360                  */
6361                 len = strlen(dp->d_name);
6362                 if (dp->d_name[len - 2 ] != 's' || dp->d_name[len - 1] != '0') {
6363                         BAM_DPRINTF((D_SKIP_SLICE_NOTZERO, fcn, dp->d_name));
6364                         continue;
6365                 }

6367                 ret = process_slice0(dp->d_name, tfp, mnttab_hash, tmpmnt);
6368                 INJECT_ERROR1("PROCESS_S0_FAIL", ret = -1);
6369                 if (ret == -1)
6370                         goto fail;
6371         }

6373         (void) closedir(dirp);
6374         free_mnttab(mnttab_hash);
6375         (void) rmdir(tmpmnt);
```



```
6377         ret = fclose(tfp);
6378         error = errno;
6379         INJECT_ERROR1("FCLOSE_SIGNLIST_TMP", ret = EOF);
6380         if (ret == EOF) {
6381                 bam_error(CLOSE_FAIL, UFS_SIGNATURE_LIST".tmp",
6382                     strerror(error));
6383                 (void) unlink(UFS_SIGNATURE_LIST".tmp");
6384                 return (-1);
6385         }

6387         /* We have a list of existing GRUB signatures. Sort it first */
6388         (void) snprintf(cmd, sizeof (cmd),
6389             "/usr/bin/sort -u %s.tmp > %s.sorted",
6390             UFS_SIGNATURE_LIST, UFS_SIGNATURE_LIST);

6392         ret = exec_cmd(cmd, NULL);
6393         INJECT_ERROR1("SORT_SIGN_LIST", ret = 1);
6394         if (ret != 0) {
6395                 bam_error(GRUBSIGN_SORT_FAILED);
6396                 (void) unlink(UFS_SIGNATURE_LIST".sorted");
6397                 (void) unlink(UFS_SIGNATURE_LIST".tmp");
6398                 return (-1);
6399         }

6401         (void) unlink(UFS_SIGNATURE_LIST".tmp");

6403         ret = rename(UFS_SIGNATURE_LIST".sorted", UFS_SIGNATURE_LIST);
6404         error = errno;
6405         INJECT_ERROR1("RENAME_TMP_SIGNLIST", ret = -1);
6406         if (ret == -1) {
6407                 bam_error(RENAME_FAIL, UFS_SIGNATURE_LIST, strerror(error));
6408                 (void) unlink(UFS_SIGNATURE_LIST".sorted");
6409                 return (-1);
6410         }

6412         if (stat(UFS_SIGNATURE_LIST, &sb) == 0 && sb.st_size == 0) {
6413                 BAM_DPRINTF((D_ZERO_LEN_SIGNLIST, fcn, UFS_SIGNATURE_LIST));
6414         }

6416         BAM_DPRINTF((D_RETURN_SUCCESS, fcn));
6417         return (0);

6419 fail:
6420         if (dirp)
6421                 (void) closedir(dirp);
6422         free_mnttab(mnttab_hash);
6423         (void) rmdir(tmpmnt);
6424         (void) fclose(tfp);
6425         (void) unlink(UFS_SIGNATURE_LIST".tmp");
6426         BAM_DPRINTF((D_RETURN_FAILURE, fcn));
6427         return (-1);
6428 }

6430 static char *
6431 create_ufs_sign(void)
6432 {
6433         struct stat     sb;
6434         int             signnum = -1;
6435         char            tmpsign[MAXNAMELEN + 1];
6436         char            *numstr;
6437         int             i;
6438         FILE            *tfp;
6439         int             ret;
6440         int             error;
6441         const char      *fcn = "create_ufs_sign()";
```

```
6443         bam_print(SEARCHING_UFS_SIGN);

6445         ret = FindAllUfsSignatures();
6446         INJECT_ERROR1("FIND_ALL_UFS", ret = -1);
6447         if (ret == -1) {
6448                 bam_error(ERR_FIND_UFS_SIGN);
6449                 return (NULL);
6450         }

6452         /* Make sure the list exists and is owned by root */
6453         INJECT_ERROR1("SIGNLIST_NOT_CREATED",
6454             (void) unlink(UFS_SIGNATURE_LIST));
6455         if (stat(UFS_SIGNATURE_LIST, &sb) == -1 || sb.st_uid != 0) {
6456                 (void) unlink(UFS_SIGNATURE_LIST);
6457                 bam_error(UFS_SIGNATURE_LIST_MISS, UFS_SIGNATURE_LIST);
6458                 return (NULL);
6459         }

6461         if (sb.st_size == 0) {
6462                 bam_print(GRUBSIGN_UFS_NONE);
6463                 i = 0;
6464                 goto found;
6465         }

6467         /* The signature list was sorted when it was created */
6468         tfp = fopen(UFS_SIGNATURE_LIST, "r");
6469         error = errno;
6470         INJECT_ERROR1("FOPEN_SIGN_LIST", tfp = NULL);
6471         if (tfp == NULL) {
6472                 bam_error(UFS_SIGNATURE_LIST_OPENERR,
6473                     UFS_SIGNATURE_LIST, strerror(error));
6474                 (void) unlink(UFS_SIGNATURE_LIST);
6475                 return (NULL);
6476         }

6478         for (i = 0; s_fgets(tmpsign, sizeof (tmpsign), tfp); i++) {

6480                 if (strncmp(tmpsign, GRUBSIGN_UFS_PREFIX,
6481                     strlen(GRUBSIGN_UFS_PREFIX)) != 0) {
6482                         (void) fclose(tfp);
6483                         (void) unlink(UFS_SIGNATURE_LIST);
6484                         bam_error(UFS_BADSIGN, tmpsign);
6485                         return (NULL);
6486                 }
6487                 numstr = tmpsign + strlen(GRUBSIGN_UFS_PREFIX);

6489                 if (numstr[0] == '\0' || !isdigit(numstr[0])) {
6490                         (void) fclose(tfp);
6491                         (void) unlink(UFS_SIGNATURE_LIST);
6492                         bam_error(UFS_BADSIGN, tmpsign);
6493                         return (NULL);
6494                 }

6496                 signnum = atoi(numstr);
6497                 INJECT_ERROR1("NEGATIVE_SIGN", signnum = -1);
6498                 if (signnum < 0) {
6499                         (void) fclose(tfp);
6500                         (void) unlink(UFS_SIGNATURE_LIST);
6501                         bam_error(UFS_BADSIGN, tmpsign);
6502                         return (NULL);
6503                 }

6505                 if (i != signnum) {
6506                         BAM_DPRINTF((D_FOUND_HOLE_SIGNLIST, fcn, i));
6507                         break;
```



```
6508                 }
6509         }

6511         (void) fclose(tfp);

6513 found:
6514         (void) snprintf(tmpsign, sizeof (tmpsign), "rootfs%d", i);

6516         /* add the ufs signature to the /var/run list of signatures */
6517         ret = ufs_add_to_sign_list(tmpsign);
6518         INJECT_ERROR1("UFS_ADD_TO_SIGN_LIST", ret = -1);
6519         if (ret == -1) {
6520                 (void) unlink(UFS_SIGNATURE_LIST);
6521                 bam_error(FAILED_ADD_SIGNLIST, tmpsign);
6522                 return (NULL);
6523         }

6525         BAM_DPRINTF((D_RETURN_SUCCESS, fcn));

6527         return (s_strdup(tmpsign));
6528 }

6530 static char *
6531 get_fstype(char *osroot)
6532 {
6533         FILE            *mntfp;
6534         struct mnttab   mp = {0};
6535         struct mnttab   mpref = {0};
6536         int             error;
6537         int             ret;
6538         const char      *fcn = "get_fstype()";

6540         INJECT_ERROR1("GET_FSTYPE_OSROOT", osroot = NULL);
6541         if (osroot == NULL) {
6542                 bam_error(GET_FSTYPE_ARGS);
6543                 return (NULL);
6544         }

6546         mntfp = fopen(MNTTAB, "r");
6547         error = errno;
6548         INJECT_ERROR1("GET_FSTYPE_FOPEN", mntfp = NULL);
6549         if (mntfp == NULL) {
6550                 bam_error(OPEN_FAIL, MNTTAB, strerror(error));
6551                 return (NULL);
6552         }

6554         if (*osroot == '\0')
6555                 mpref.mnt_mountp = "/";
6556         else
6557                 mpref.mnt_mountp = osroot;

6559         ret = getmntany(mntfp, &mp, &mpref);
6560         INJECT_ERROR1("GET_FSTYPE_GETMNTANY", ret = 1);
6561         if (ret != 0) {
6562                 bam_error(MNTTAB_MNTPT_NOT_FOUND, osroot, MNTTAB);
6563                 (void) fclose(mntfp);
6564                 return (NULL);
6565         }
6566         (void) fclose(mntfp);

6568         INJECT_ERROR1("GET_FSTYPE_NULL", mp.mnt_fstype = NULL);
6569         if (mp.mnt_fstype == NULL) {
6570                 bam_error(MNTTAB_FSTYPE_NULL, osroot);
6571                 return (NULL);
6572         }
```

```
6574         BAM_DPRINTF((D_RETURN_SUCCESS, fcn));

6576         return (s_strdup(mp.mnt_fstype));
6577 }

6579 static char *
6580 create_zfs_sign(char *osdev)
6581 {
6582         char            tmpsign[PATH_MAX];
6583         char            *pool;
6584         const char      *fcn = "create_zfs_sign()";

6586         BAM_DPRINTF((D_FUNC_ENTRY1, fcn, osdev));

6588         /*
6589          * First find the pool name
6590          */
6591         pool = get_pool(osdev);
6592         INJECT_ERROR1("CREATE_ZFS_SIGN_GET_POOL", pool = NULL);
6593         if (pool == NULL) {
6594                 bam_error(GET_POOL_FAILED, osdev);
6595                 return (NULL);
6596         }

6598         (void) snprintf(tmpsign, sizeof (tmpsign), "pool_%s", pool);

6600         BAM_DPRINTF((D_CREATED_ZFS_SIGN, fcn, tmpsign));

6602         free(pool);

6604         BAM_DPRINTF((D_RETURN_SUCCESS, fcn));

6606         return (s_strdup(tmpsign));
6607 }

6609 static char *
6610 create_new_sign(char *osdev, char *fstype)
6611 {
6612         char            *sign;
6613         const char      *fcn = "create_new_sign()";

6615         INJECT_ERROR1("NEW_SIGN_FSTYPE", fstype = "foofs");

6617         if (strcmp(fstype, "zfs") == 0) {
6618                 BAM_DPRINTF((D_CREATE_NEW_ZFS, fcn));
6619                 sign = create_zfs_sign(osdev);
6620         } else if (strcmp(fstype, "ufs") == 0) {
6621                 BAM_DPRINTF((D_CREATE_NEW_UFS, fcn));
6622                 sign = create_ufs_sign();
6623         } else {
6624                 bam_error(GRUBSIGN_NOTSUP, fstype);
6625                 sign = NULL;
6626         }

6628         BAM_DPRINTF((D_CREATED_NEW_SIGN, fcn, sign ? sign : "<NULL>"));
6629         return (sign);
6630 }

6632 static int
6633 set_backup_common(char *mntpt, char *sign)
6634 {
6635         FILE            *bfp;
6636         char            backup[PATH_MAX];
6637         char            tmpsign[PATH_MAX];
6638         int             error;
6639         char            *bdir;
```



```
6640         char            *backup_dup;
6641         struct stat     sb;
6642         int             ret;
6643         const char      *fcn = "set_backup_common()";

6645         (void) snprintf(backup, sizeof (backup), "%s%s",
6646             mntpt, GRUBSIGN_BACKUP);

6648         /* First read the backup */
6649         bfp = fopen(backup, "r");
6650         if (bfp != NULL) {
6651                 while (s_fgets(tmpsign, sizeof (tmpsign), bfp)) {
6652                         if (strcmp(tmpsign, sign) == 0) {
6653                                 BAM_DPRINTF((D_FOUND_IN_BACKUP, fcn, sign));
6654                                 (void) fclose(bfp);
6655                                 return (0);
6656                         }
6657                 }
6658                 (void) fclose(bfp);
6659                 BAM_DPRINTF((D_NOT_FOUND_IN_EXIST_BACKUP, fcn, sign));
6660         } else {
6661                 BAM_DPRINTF((D_BACKUP_NOT_EXIST, fcn, backup));
6662         }

6664         /*
6665          * Didn't find the correct signature. First create
6666          * the directory if necessary.
6667          */

6669         /* dirname() modifies its argument so dup it */
6670         backup_dup = s_strdup(backup);
6671         bdir = dirname(backup_dup);
6672         assert(bdir);

6674         ret = stat(bdir, &sb);
6675         INJECT_ERROR1("SET_BACKUP_STAT", ret = -1);
6676         if (ret == -1) {
6677                 BAM_DPRINTF((D_BACKUP_DIR_NOEXIST, fcn, bdir));
6678                 ret = mkdirp(bdir, DIR_PERMS);
6679                 error = errno;
6680                 INJECT_ERROR1("SET_BACKUP_MKDIRP", ret = -1);
6681                 if (ret == -1) {
6682                         bam_error(GRUBSIGN_BACKUP_MKDIRERR,
6683                             GRUBSIGN_BACKUP, strerror(error));
6684                         free(backup_dup);
6685                         return (-1);
6686                 }
6687         }
6688         free(backup_dup);

6690         /*
6691          * Open the backup in append mode to add the correct
6692          * signature;
6693          */
6694         bfp = fopen(backup, "a");
6695         error = errno;
6696         INJECT_ERROR1("SET_BACKUP_FOPEN_A", bfp = NULL);
6697         if (bfp == NULL) {
6698                 bam_error(GRUBSIGN_BACKUP_OPENERR,
6699                     GRUBSIGN_BACKUP, strerror(error));
6700                 return (-1);
6701         }

6703         (void) snprintf(tmpsign, sizeof (tmpsign), "%s\n", sign);

6705         ret = fputs(tmpsign, bfp);
```

```
6706         error = errno;
6707         INJECT_ERROR1("SET_BACKUP_FPUTS", ret = 0);
6708         if (ret != strlen(tmpsign)) {
6709                 bam_error(GRUBSIGN_BACKUP_WRITEERR,
6710                     GRUBSIGN_BACKUP, strerror(error));
6711                 (void) fclose(bfp);
6712                 return (-1);
6713         }

6715         (void) fclose(bfp);

6717         if (bam_verbose)
6718                 bam_print(GRUBSIGN_BACKUP_UPDATED, GRUBSIGN_BACKUP);

6720         BAM_DPRINTF((D_RETURN_SUCCESS, fcn));

6722         return (0);
6723 }

6725 static int
6726 set_backup_ufs(char *osroot, char *sign)
6727 {
6728         const char      *fcn = "set_backup_ufs()";

6730         BAM_DPRINTF((D_FUNC_ENTRY2, fcn, osroot, sign));
6731         return (set_backup_common(osroot, sign));
6732 }

6734 static int
6735 set_backup_zfs(char *osdev, char *sign)
6736 {
6737         char            *pool;
6738         char            *mntpt;
6739         zfs_mnted_t     mnted;
6740         int             ret;
6741         const char      *fcn = "set_backup_zfs()";

6743         BAM_DPRINTF((D_FUNC_ENTRY2, fcn, osdev, sign));

6745         pool = get_pool(osdev);
6746         INJECT_ERROR1("SET_BACKUP_GET_POOL", pool = NULL);
6747         if (pool == NULL) {
6748                 bam_error(GET_POOL_FAILED, osdev);
6749                 return (-1);
6750         }

6752         mntpt = mount_top_dataset(pool, &mnted);
6753         INJECT_ERROR1("SET_BACKUP_MOUNT_DATASET", mntpt = NULL);
6754         if (mntpt == NULL) {
6755                 bam_error(FAIL_MNT_TOP_DATASET, pool);
6756                 free(pool);
6757                 return (-1);
6758         }

6760         ret = set_backup_common(mntpt, sign);

6762         (void) umount_top_dataset(pool, mnted, mntpt);

6764         free(pool);

6766         INJECT_ERROR1("SET_BACKUP_ZFS_FAIL", ret = 1);
6767         if (ret == 0) {
6768                 BAM_DPRINTF((D_RETURN_SUCCESS, fcn));
6769         } else {
6770                 BAM_DPRINTF((D_RETURN_FAILURE, fcn));
6771         }
```



```
6773         return (ret);
6774 }

6776 static int
6777 set_backup(char *osroot, char *osdev, char *sign, char *fstype)
6778 {
6779         const char      *fcn = "set_backup()";
6780         int             ret;

6782         INJECT_ERROR1("SET_BACKUP_FSTYPE", fstype = "foofs");

6784         if (strcmp(fstype, "ufs") == 0) {
6785                 BAM_DPRINTF((D_SET_BACKUP_UFS, fcn));
6786                 ret = set_backup_ufs(osroot, sign);
6787         } else if (strcmp(fstype, "zfs") == 0) {
6788                 BAM_DPRINTF((D_SET_BACKUP_ZFS, fcn));
6789                 ret = set_backup_zfs(osdev, sign);
6790         } else {
6791                 bam_error(GRUBSIGN_NOTSUP, fstype);
6792                 ret = -1;
6793         }

6795         if (ret == 0) {
6796                 BAM_DPRINTF((D_RETURN_SUCCESS, fcn));
6797         } else {
6798                 BAM_DPRINTF((D_RETURN_FAILURE, fcn));
6799         }

6801         return (ret);
6802 }

6804 static int
6805 set_primary_common(char *mntpt, char *sign)
6806 {
6807         char            signfile[PATH_MAX];
6808         char            signdir[PATH_MAX];
6809         struct stat     sb;
6810         int             fd;
6811         int             error;
6812         int             ret;
6813         const char      *fcn = "set_primary_common()";

6815         (void) snprintf(signfile, sizeof (signfile), "%s/%s/%s",
6816             mntpt, GRUBSIGN_DIR, sign);

6818         if (stat(signfile, &sb) != -1) {
6819                 if (bam_verbose)
6820                         bam_print(PRIMARY_SIGN_EXISTS, sign);
6821                 return (0);
6822         } else {
6823                 BAM_DPRINTF((D_PRIMARY_NOT_EXIST, fcn, signfile));
6824         }

6826         (void) snprintf(signdir, sizeof (signdir), "%s/%s",
6827             mntpt, GRUBSIGN_DIR);

6829         if (stat(signdir, &sb) == -1) {
6830                 BAM_DPRINTF((D_PRIMARY_DIR_NOEXIST, fcn, signdir));
6831                 ret = mkdirp(signdir, DIR_PERMS);
6832                 error = errno;
6833                 INJECT_ERROR1("SET_PRIMARY_MKDIRP", ret = -1);
6834                 if (ret == -1) {
6835                         bam_error(GRUBSIGN_MKDIR_ERR, signdir, strerror(errno));
6836                         return (-1);
6837                 }
```

```
6838         }

6840         fd = open(signfile, O_RDWR|O_CREAT|O_TRUNC, 0444);
6841         error = errno;
6842         INJECT_ERROR1("PRIMARY_SIGN_CREAT", fd = -1);
6843         if (fd == -1) {
6844                 bam_error(GRUBSIGN_PRIMARY_CREATERR, signfile, strerror(error));
6845                 return (-1);
6846         }

6848         ret = fsync(fd);
6849         error = errno;
6850         INJECT_ERROR1("PRIMARY_FSYNC", ret = -1);
6851         if (ret != 0) {
6852                 bam_error(GRUBSIGN_PRIMARY_SYNCERR, signfile, strerror(error));
6853         }

6855         (void) close(fd);

6857         if (bam_verbose)
6858                 bam_print(GRUBSIGN_CREATED_PRIMARY, signfile);

6860         BAM_DPRINTF((D_RETURN_SUCCESS, fcn));

6862         return (0);
6863 }

6865 static int
6866 set_primary_ufs(char *osroot, char *sign)
6867 {
6868         const char      *fcn = "set_primary_ufs()";

6870         BAM_DPRINTF((D_FUNC_ENTRY2, fcn, osroot, sign));
6871         return (set_primary_common(osroot, sign));
6872 }

6874 static int
6875 set_primary_zfs(char *osdev, char *sign)
6876 {
6877         char            *pool;
6878         char            *mntpt;
6879         zfs_mnted_t     mnted;
6880         int             ret;
6881         const char      *fcn = "set_primary_zfs()";

6883         BAM_DPRINTF((D_FUNC_ENTRY2, fcn, osdev, sign));

6885         pool = get_pool(osdev);
6886         INJECT_ERROR1("SET_PRIMARY_ZFS_GET_POOL", pool = NULL);
6887         if (pool == NULL) {
6888                 bam_error(GET_POOL_FAILED, osdev);
6889                 return (-1);
6890         }

6892         /* Pool name must exist in the sign */
6893         ret = (strstr(sign, pool) != NULL);
6894         INJECT_ERROR1("SET_PRIMARY_ZFS_POOL_SIGN_INCOMPAT", ret = 0);
6895         if (ret == 0) {
6896                 bam_error(POOL_SIGN_INCOMPAT, pool, sign);
6897                 free(pool);
6898                 return (-1);
6899         }

6901         mntpt = mount_top_dataset(pool, &mnted);
6902         INJECT_ERROR1("SET_PRIMARY_ZFS_MOUNT_DATASET", mntpt = NULL);
6903         if (mntpt == NULL) {
```



```
6904                 bam_error(FAIL_MNT_TOP_DATASET, pool);
6905                 free(pool);
6906                 return (-1);
6907         }

6909         ret = set_primary_common(mntpt, sign);

6911         (void) umount_top_dataset(pool, mnted, mntpt);

6913         free(pool);

6915         INJECT_ERROR1("SET_PRIMARY_ZFS_FAIL", ret = 1);
6916         if (ret == 0) {
6917                 BAM_DPRINTF((D_RETURN_SUCCESS, fcn));
6918         } else {
6919                 BAM_DPRINTF((D_RETURN_FAILURE, fcn));
6920         }

6922         return (ret);
6923 }

6925 static int
6926 set_primary(char *osroot, char *osdev, char *sign, char *fstype)
6927 {
6928         const char      *fcn = "set_primary()";
6929         int             ret;

6931         INJECT_ERROR1("SET_PRIMARY_FSTYPE", fstype = "foofs");
6932         if (strcmp(fstype, "ufs") == 0) {
6933                 BAM_DPRINTF((D_SET_PRIMARY_UFS, fcn));
6934                 ret = set_primary_ufs(osroot, sign);
6935         } else if (strcmp(fstype, "zfs") == 0) {
6936                 BAM_DPRINTF((D_SET_PRIMARY_ZFS, fcn));
6937                 ret = set_primary_zfs(osdev, sign);
6938         } else {
6939                 bam_error(GRUBSIGN_NOTSUP, fstype);
6940                 ret = -1;
6941         }

6943         if (ret == 0) {
6944                 BAM_DPRINTF((D_RETURN_SUCCESS, fcn));
6945         } else {
6946                 BAM_DPRINTF((D_RETURN_FAILURE, fcn));
6947         }

6949         return (ret);
6950 }

6952 static int
6953 ufs_add_to_sign_list(char *sign)
6954 {
6955         FILE            *tfp;
6956         char            signline[MAXNAMELEN];
6957         char            cmd[PATH_MAX];
6958         int             ret;
6959         int             error;
6960         const char      *fcn = "ufs_add_to_sign_list()";

6962         INJECT_ERROR1("ADD_TO_SIGN_LIST_NOT_UFS", sign = "pool_rpool5");
6963         if (strncmp(sign, GRUBSIGN_UFS_PREFIX,
6964             strlen(GRUBSIGN_UFS_PREFIX)) != 0) {
6965                 bam_error(INVALID_UFS_SIGN, sign);
6966                 (void) unlink(UFS_SIGNATURE_LIST);
6967                 return (-1);
6968         }
```

```
6970         /*
6971          * most failures in this routine are not a fatal error
6972          * We simply unlink the /var/run file and continue
6973          */
6975         ret = rename(UFS_SIGNATURE_LIST, UFS_SIGNATURE_LIST".tmp");
6976         error = errno;
6977         INJECT_ERROR1("ADD_TO_SIGN_LIST_RENAME", ret = -1);
6978         if (ret == -1) {
6979                 bam_error(RENAME_FAIL, UFS_SIGNATURE_LIST".tmp",
6980                     strerror(error));
6981                 (void) unlink(UFS_SIGNATURE_LIST);
6982                 return (0);
6983         }
6985         tfp = fopen(UFS_SIGNATURE_LIST".tmp", "a");
6986         error = errno;
6987         INJECT_ERROR1("ADD_TO_SIGN_LIST_FOPEN", tfp = NULL);
6988         if (tfp == NULL) {
6989                 bam_error(OPEN_FAIL, UFS_SIGNATURE_LIST".tmp", strerror(error));
6990                 (void) unlink(UFS_SIGNATURE_LIST".tmp");
6991                 return (0);
6992         }
6994         (void) snprintf(signline, sizeof (signline), "%s\n", sign);
6996         ret = fputs(signline, tfp);
6997         error = errno;
6998         INJECT_ERROR1("ADD_TO_SIGN_LIST_FPUTS", ret = 0);
6999         if (ret != strlen(signline)) {
7000                 bam_error(SIGN_LIST_FPUTS_ERR, sign, strerror(error));
7001                 (void) fclose(tfp);
7002                 (void) unlink(UFS_SIGNATURE_LIST".tmp");
7003                 return (0);
7004         }
7006         ret = fclose(tfp);
7007         error = errno;
7008         INJECT_ERROR1("ADD_TO_SIGN_LIST_FCLOSE", ret = EOF);
7009         if (ret == EOF) {
7010                 bam_error(CLOSE_FAIL, UFS_SIGNATURE_LIST".tmp",
7011                     strerror(error));
7012                 (void) unlink(UFS_SIGNATURE_LIST".tmp");
7013                 return (0);
7014         }
7016         /* Sort the list again */
7017         (void) snprintf(cmd, sizeof (cmd),
7018             "/usr/bin/sort -u %s.tmp > %s.sorted",
7019             UFS_SIGNATURE_LIST, UFS_SIGNATURE_LIST);
7021         ret = exec_cmd(cmd, NULL);
7022         INJECT_ERROR1("ADD_TO_SIGN_LIST_SORT", ret = 1);
7023         if (ret != 0) {
7024                 bam_error(GRUBSIGN_SORT_FAILED);
7025                 (void) unlink(UFS_SIGNATURE_LIST".sorted");
7026                 (void) unlink(UFS_SIGNATURE_LIST".tmp");
7027                 return (0);
7028         }
7030         (void) unlink(UFS_SIGNATURE_LIST".tmp");
7032         ret = rename(UFS_SIGNATURE_LIST".sorted", UFS_SIGNATURE_LIST);
7033         error = errno;
7034         INJECT_ERROR1("ADD_TO_SIGN_LIST_RENAME2", ret = -1);
7035         if (ret == -1) {
```



```
7036                 bam_error(RENAME_FAIL, UFS_SIGNATURE_LIST, strerror(error));
7037                 (void) unlink(UFS_SIGNATURE_LIST".sorted");
7038                 return (0);
7039         }
7041         BAM_DPRINTF((D_RETURN_SUCCESS, fcn));
7043         return (0);
7044 }
7046 static int
7047 set_signature(char *osroot, char *osdev, char *sign, char *fstype)
7048 {
7049         int             ret;
7050         const char      *fcn = "set_signature()";
7052         BAM_DPRINTF((D_FUNC_ENTRY4, fcn, osroot, osdev, sign, fstype));
7054         ret = set_backup(osroot, osdev, sign, fstype);
7055         INJECT_ERROR1("SET_SIGNATURE_BACKUP", ret = -1);
7056         if (ret == -1) {
7057                 BAM_DPRINTF((D_RETURN_FAILURE, fcn));
7058                 bam_error(SET_BACKUP_FAILED, sign, osroot, osdev);
7059                 return (-1);
7060         }
7062         ret = set_primary(osroot, osdev, sign, fstype);
7063         INJECT_ERROR1("SET_SIGNATURE_PRIMARY", ret = -1);
7065         if (ret == 0) {
7066                 BAM_DPRINTF((D_RETURN_SUCCESS, fcn));
7067         } else {
7068                 BAM_DPRINTF((D_RETURN_FAILURE, fcn));
7069                 bam_error(SET_PRIMARY_FAILED, sign, osroot, osdev);
7071         }
7072         return (ret);
7073 }
7075 char *
7076 get_grubsign(char *osroot, char *osdev)
7077 {
7078         char            *grubsign;      /* (<sign>,#,#) */
7079         char            *slice;
7080         int             fdiskpart;
7081         char            *sign;
7082         char            *fstype;
7083         int             ret;
7084         const char      *fcn = "get_grubsign()";
7086         BAM_DPRINTF((D_FUNC_ENTRY2, fcn, osroot, osdev));
7087         fstype = get_fstype(osroot);
7088         INJECT_ERROR1("GET_GRUBSIGN_FSTYPE", fstype = NULL);
7089         if (fstype == NULL) {
7090                 bam_error(GET_FSTYPE_FAILED, osroot);
7091                 return (NULL);
7092         }
7094         sign = find_existing_sign(osroot, osdev, fstype);
7095         INJECT_ERROR1("FIND_EXISTING_SIGN", sign = NULL);
7096         if (sign == NULL) {
7097                 BAM_DPRINTF((D_GET_GRUBSIGN_NO_EXISTING, fcn, osroot, osdev));
7098                 sign = create_new_sign(osdev, fstype);
7099                 INJECT_ERROR1("CREATE_NEW_SIGN", sign = NULL);
7100                 if (sign == NULL) {
7101                         bam_error(GRUBSIGN_CREATE_FAIL, osdev);
```

```
7102                        free(fstype);
7103                        return (NULL);
7104                }
7105        }

7107        ret = set_signature(osroot, osdev, sign, fstype);
7108        INJECT_ERROR1("SET_SIGNATURE_FAIL", ret = -1);
7109        if (ret == -1) {
7110                bam_error(GRUBSIGN_WRITE_FAIL, osdev);
7111                free(sign);
7112                free(fstype);
7113                (void) unlink(UFS_SIGNATURE_LIST);
7114                return (NULL);
7115        }

7117        free(fstype);

7119        if (bam_verbose)
7120                bam_print(GRUBSIGN_FOUND_OR_CREATED, sign, osdev);

7122        fdiskpart = get_partition(osdev);
7123        INJECT_ERROR1("GET_GRUBSIGN_FDISK", fdiskpart = -1);
7124        if (fdiskpart == -1) {
7125                bam_error(FDISKPART_FAIL, osdev);
7126                free(sign);
7127                return (NULL);
7128        }

7130        slice = strrchr(osdev, 's');

7132        grubsign = s_calloc(1, MAXNAMELEN + 10);
7133 /*     if (slice) {
4598        if (slice) {
7134                (void) snprintf(grubsign, MAXNAMELEN + 10, "(%s,%d,%c)",
7135                    sign, fdiskpart, slice[1] + 'a' - '0');
7136        } else
7137                (void) snprintf(grubsign, MAXNAMELEN + 10, "(%s,%d)",
7138                    sign, fdiskpart);*/
7139        grubsign = strdup(sign);
4603                    sign, fdiskpart);

7141        free(sign);

7143        BAM_DPRINTF((D_GET_GRUBSIGN_SUCCESS, fcn, grubsign));

7145        return (strchr(grubsign,'_') + 1);
4609        return (grubsign);
7146 }
_____unchanged_portion_omitted_

7771 /*
7772  * look for matching bootadm entry with specified parameters
7773  * Here are the rules (based on existing usage):
7774  * - If title is specified, match on title only
7775  * - Else, match on root/findroot, kernel, and module.
7776  *   Note that, if root_opt is non-zero, the absence of
7777  *   root line is considered a match.
7778  */
7779 static entry_t *
7780 find_boot_entry(
7781         menu_t *mp,
7782         char *title,
7783         char *kernel,
7784         char *findroot,
7785         char *root,
7786         char *module,
```



```
7787         int root_opt,
7788         int *entry_num)
7789 {
7790         int             i;
7791         line_t          *lp;
7792         entry_t         *ent;
7793         const char      *fcn = "find_boot_entry()";

7795         if (entry_num)
7796                 *entry_num = BAM_ERROR;

7798         /* find matching entry */
7799         for (i = 0, ent = mp->entries; ent; i++, ent = ent->next) {
7800                 lp = ent->start;

7802                 /* first line of entry must be bootadm comment */
7803                 lp = ent->start;
7804                 if (lp->flags != BAM_COMMENT ||
7805                     strcmp(lp->arg, BAM_BOOTADM_HDR) != 0) {
7806                         continue;
7807                 }

7809                 /* advance to title line */
7810                 lp = lp->next;
7811                 if (title) {
7812                         if (lp->flags == BAM_TITLE && lp->arg &&
7813                             strcmp(lp->arg, title) == 0) {
7814                                 BAM_DPRINTF((D_MATCHED_TITLE, fcn, title));
7815                                 break;
7816                         }
7817                         BAM_DPRINTF((D_NOMATCH_TITLE, fcn, title, lp->arg));
7818                         continue;       /* check title only */
7819                 }

7821                 lp = lp->next;  /* advance to root line */
7822                 if (lp == NULL) {
7823                         continue;
7824                 } else if (lp->cmd != NULL &&
7825                     strcmp(lp->cmd, menu_cmds[FINDROOT_CMD]) == 0) {
7826                         INJECT_ERROR1("FIND_BOOT_ENTRY_NULL_FINDROOT",
7827                             findroot = NULL);
7828                         if (findroot == NULL) {
7829                                 BAM_DPRINTF((D_NOMATCH_FINDROOT_NULL,
7830                                     fcn, lp->arg));
7831                                 continue;
7832                         }
7833                         /* findroot command found, try match  */
7834                         if (strncmp(lp->arg, strchr(findroot, '_') + 1, strlen(l
5298                         if (strcmp(lp->arg, findroot) != 0) {
7835                                 BAM_DPRINTF((D_NOMATCH_FINDROOT,
7836                                     fcn, findroot, lp->arg));
7837                                 continue;
7838                         }
7839                         BAM_DPRINTF((D_MATCHED_FINDROOT, fcn, findroot));
7840                         lp = lp->next;  /* advance to kernel line */
7841                 } else if (lp->cmd != NULL &&
7842                     strcmp(lp->cmd, menu_cmds[ROOT_CMD]) == 0) {
7843                         INJECT_ERROR1("FIND_BOOT_ENTRY_NULL_ROOT", root = NULL);
7844                         if (root == NULL) {
7845                                 BAM_DPRINTF((D_NOMATCH_ROOT_NULL,
7846                                     fcn, lp->arg));
7847                                 continue;
7848                         }
7849                         /* root cmd found, try match */
7850                         if (strcmp(lp->arg, root) != 0) {
7851                                 BAM_DPRINTF((D_NOMATCH_ROOT,
```

```
7852                                    fcn, root, lp->arg));
7853                                continue;
7854                        }
7855                        BAM_DPRINTF((D_MATCHED_ROOT, fcn, root));
7856                        lp = lp->next;  /* advance to kernel line */
7857                } else {
7858                        INJECT_ERROR1("FIND_BOOT_ENTRY_ROOT_OPT_NO",
7859                            root_opt = 0);
7860                        INJECT_ERROR1("FIND_BOOT_ENTRY_ROOT_OPT_YES",
7861                            root_opt = 1);
7862                        /* no root command, see if root is optional */
7863                        if (root_opt == 0) {
7864                                BAM_DPRINTF((D_NO_ROOT_OPT, fcn));
7865                                continue;
7866                        }
7867                        BAM_DPRINTF((D_ROOT_OPT, fcn));
7868                }

7870                if (lp == NULL || lp->next == NULL) {
7871                        continue;
7872                }

7874                if (kernel &&
7875                    (!check_cmd(lp->cmd, KERNEL_CMD, lp->arg, kernel))) {
7876                        if (!(ent->flags & BAM_ENTRY_FAILSAFE) ||
7877                            !(ent->flags & BAM_ENTRY_DBOOT) ||
7878                            strcmp(kernel, DIRECT_BOOT_FAILSAFE_LINE) != 0)
7879                                continue;

7881                        ent->flags |= BAM_ENTRY_UPGFSKERNEL;

7883                }
7884                BAM_DPRINTF((D_KERNEL_MATCH, fcn, kernel, lp->arg));

7886                /*
7887                 * Check for matching module entry (failsafe or normal).
7888                 * If it fails to match, we go around the loop again.
7889                 * For xpv entries, there are two module lines, so we
7890                 * do the check twice.
7891                 */
7892                lp = lp->next;  /* advance to options line */
7893 #endif /* ! codereview */
7894                lp = lp->next;  /* advance to module line */
7895                if (check_cmd(lp->cmd, MODULE_CMD, lp->arg, module) ||
7896                    (((lp = lp->next) != NULL) &&
7897                    check_cmd(lp->cmd, MODULE_CMD, lp->arg, module))) {
7898                        /* match found */
7899                        BAM_DPRINTF((D_MODULE_MATCH, fcn, module, lp->arg));
7900                        break;
7901                }

7903                if (strcmp(module, FAILSAFE_ARCHIVE) == 0 &&
7904                    (strcmp(lp->prev->arg, FAILSAFE_ARCHIVE_32) == 0 ||
7905                    strcmp(lp->prev->arg, FAILSAFE_ARCHIVE_64) == 0)) {
7906                        ent->flags |= BAM_ENTRY_UPGFSMODULE;
7907                        break;
7908                }

7910        }

7912        if (ent && entry_num) {
7913                *entry_num = i;
7914        }

7916        if (ent) {
7917                BAM_DPRINTF((D_RETURN_RET, fcn, i));
```



```
7918        } else {
7919                BAM_DPRINTF((D_RETURN_RET, fcn, BAM_ERROR));
7920        }
7921        return (ent);
7922 }

7924 static int
7925 update_boot_entry(menu_t *mp, char *title, char *findroot, char *root,
7926     char *kernel, char *mod_kernel, char *module, int root_opt)
7927 {
7928        int             i;
7929        int             change_kernel = 0;
7930        entry_t         *ent;
7931        line_t          *lp;
7932        line_t          *tlp;
7933        char            linebuf[BAM_MAXLINE];
7934        const char      *fcn = "update_boot_entry()";
7935        char            *label;
7936 #endif /* ! codereview */

7938        /* note: don't match on title, it's updated on upgrade */
7939        ent = find_boot_entry(mp, NULL, kernel, findroot, root, module,
7940            root_opt, &i);
7941        if ((ent == NULL) && (bam_direct == BAM_DIRECT_DBOOT)) {
7942                /*
7943                 * We may be upgrading a kernel from multiboot to
7944                 * directboot.  Look for a multiboot entry. A multiboot
7945                 * entry will not have a findroot line.
7946                 */
7947                ent = find_boot_entry(mp, NULL, "multiboot", NULL, root,
7948                    MULTIBOOT_ARCHIVE, root_opt, &i);
7949                if (ent != NULL) {
7950                        BAM_DPRINTF((D_UPGRADE_FROM_MULTIBOOT, fcn, root));
7951                        change_kernel = 1;
7952                }
7953        } else if (ent) {
7954                BAM_DPRINTF((D_FOUND_FINDROOT, fcn, findroot));
7955        }

7957        if (ent == NULL) {
7958                BAM_DPRINTF((D_ENTRY_NOT_FOUND_CREATING, fcn, findroot));
7959                return (add_boot_entry(mp, title, findroot,
7960                    kernel, mod_kernel, module, NULL));
7961        }

7963        /* replace title of existing entry and update findroot line */
7964        lp = ent->start;
7965        lp = lp->next;  /* title line */
7966        (void) snprintf(linebuf, sizeof (linebuf), "%s%s%s",
7967            menu_cmds[TITLE_CMD], menu_cmds[SEP_CMD], title);
7968        free(lp->arg);
7969        free(lp->line);
7970        lp->arg = s_strdup(title);
7971        lp->line = s_strdup(linebuf);
7972        BAM_DPRINTF((D_CHANGING_TITLE, fcn, title));

7974        tlp = lp;       /* title line */
7975        lp = lp->next;  /* root line */

7977        /* if no root or findroot command, create a new line_t */
7978        if ((lp->cmd != NULL) && (strcmp(lp->cmd, menu_cmds[ROOT_CMD]) != 0 &&
7979            strcmp(lp->cmd, menu_cmds[FINDROOT_CMD]) != 0)) {
7980                lp = s_calloc(1, sizeof (line_t));
7981                bam_add_line(mp, ent, tlp, lp);
7982        } else {
7983                if (lp->cmd != NULL)
```

```
7984                        free(lp->cmd);

7986                free(lp->sep);
7987                free(lp->arg);
7988                free(lp->line);
7989        }

7991        lp->cmd = s_strdup(menu_cmds[FINDROOT_CMD]);
7992        lp->sep = s_strdup(menu_cmds[SEP_CMD]);
7993        label = s_strdup(strchr(findroot, '_') + 1);
7994        *(strchr(label,',')) = 0;
7995        lp->arg = s_strdup(label);
5356        lp->arg = s_strdup(findroot);
7996        (void) snprintf(linebuf, sizeof (linebuf), "%s%s%s",
7997            menu_cmds[FINDROOT_CMD], menu_cmds[SEP_CMD], label);
5358            menu_cmds[FINDROOT_CMD], menu_cmds[SEP_CMD], findroot);
7998        lp->line = s_strdup(linebuf);
7999        free(label);
8000 #endif /* ! codereview */
8001        BAM_DPRINTF((D_ADDING_FINDROOT_LINE, fcn, findroot));

8003        /* kernel line */
8004        lp = lp->next;

8006        if (ent->flags & BAM_ENTRY_UPGFSKERNEL) {
8007                char            *params = NULL;
8008                char            *opts = NULL;
8009 #endif /* ! codereview */

8011                opts = strpbrk(kernel, " \t");
8012                *opts++ = '\0';
5360                params = strstr(lp->line, "-s");
5361                if (params != NULL)
5362                        (void) snprintf(linebuf, sizeof (linebuf), "%s%s%s%s",
5363                            menu_cmds[KERNEL_DOLLAR_CMD], menu_cmds[SEP_CMD],
5364                            kernel, params+2);
5365                else
8013                (void) snprintf(linebuf, sizeof (linebuf), "%s%s%s",
8014                    menu_cmds[KERNEL_DOLLAR_CMD], menu_cmds[SEP_CMD],
8015                    kernel);

8017                if (lp->cmd != NULL)
8018                        free(lp->cmd);

8020                free(lp->arg);
8021                free(lp->line);
8022                lp->cmd = s_strdup(menu_cmds[KERNEL_DOLLAR_CMD]);
8023                lp->arg = s_strdup(strstr(linebuf, "/"));
8024                lp->line = s_strdup(linebuf);
8025                ent->flags &= ~BAM_ENTRY_UPGFSKERNEL;
8026                BAM_DPRINTF((D_ADDING_KERNEL_DOLLAR, fcn, lp->prev->cmd));

8028                lp = lp->next;
8029                params = strstr(lp->arg, "-s");
8030                free(lp->arg);
8031                free(lp->line);
8032                if (params)
8033                        (void) snprintf(linebuf, sizeof(linebuf), "%s%s%s -s",
8034                                lp->cmd, menu_cmds[SEP_CMD],opts);
8035                else
8036                        (void) snprintf(linebuf, sizeof(linebuf), "%s%s%s",
8037                                lp->cmd, menu_cmds[SEP_CMD],opts);
8038                lp->line = s_strdup(linebuf);
8039                lp->arg = s_strdup(strchr(linebuf, '=') + 1);
8040 #endif /* ! codereview */
8041        }
```



```
8043        if (change_kernel) {
8044                char            *opts = NULL;
8045 #endif /* ! codereview */
8046                /*
8047                 * We're upgrading from multiboot to directboot.
8048                 */
8049                opts = strpbrk(kernel, " \t");
8050                *opts++ = '\0';
8051 #endif /* ! codereview */
8052                if (lp->cmd != NULL &&
8053                    strcmp(lp->cmd, menu_cmds[KERNEL_CMD]) == 0) {
8054                        (void) snprintf(linebuf, sizeof (linebuf), "%s%s%s",
8055                            menu_cmds[KERNEL_DOLLAR_CMD], menu_cmds[SEP_CMD],
8056                            kernel);
8057                        free(lp->cmd);
8058                        free(lp->arg);
8059                        free(lp->line);
8060                        lp->cmd = s_strdup(menu_cmds[KERNEL_DOLLAR_CMD]);
8061                        lp->arg = s_strdup(kernel);
8062                        lp->line = s_strdup(linebuf);
8063                        lp = lp->next;
8064                        BAM_DPRINTF((D_ADDING_KERNEL_DOLLAR, fcn, kernel));
8065                        (void) snprintf(linebuf, sizeof(linebuf), "%s%s%s",
8066                                lp->cmd, menu_cmds[SEP_CMD],opts);
8067                        free(lp->arg);
8068                        free(lp->line);
8069                        lp->line = s_strdup(linebuf);
8070                        lp->arg = s_strdup(strchr(linebuf, '=') + 1);
8071 #endif /* ! codereview */
8072                }
8073                if (lp->cmd != NULL &&
8074                    strcmp(lp->cmd, menu_cmds[MODULE_CMD]) == 0) {
8075                        (void) snprintf(linebuf, sizeof (linebuf), "%s%s%s",
8076                            menu_cmds[MODULE_DOLLAR_CMD], menu_cmds[SEP_CMD],
8077                            module);
8078                        free(lp->cmd);
8079                        free(lp->arg);
8080                        free(lp->line);
8081                        lp->cmd = s_strdup(menu_cmds[MODULE_DOLLAR_CMD]);
8082                        lp->arg = s_strdup(module);
8083                        lp->line = s_strdup(linebuf);
8084                        lp = lp->next;
8085                        BAM_DPRINTF((D_ADDING_MODULE_DOLLAR, fcn, module));
8086                }
8087        }

8089        /* module line */
8090        lp = lp->next;

8092        if (ent->flags & BAM_ENTRY_UPGFSMODULE) {
8093                if (lp->cmd != NULL &&
8094                    strcmp(lp->cmd, menu_cmds[MODULE_CMD]) == 0) {
8095                        (void) snprintf(linebuf, sizeof (linebuf), "%s%s%s",
8096                            menu_cmds[MODULE_DOLLAR_CMD], menu_cmds[SEP_CMD],
8097                            module);
8098                        free(lp->cmd);
8099                        free(lp->arg);
8100                        free(lp->line);
8101                        lp->cmd = s_strdup(menu_cmds[MODULE_DOLLAR_CMD]);
8102                        lp->arg = s_strdup(module);
8103                        lp->line = s_strdup(linebuf);
8104                        lp = lp->next;
8105                        ent->flags &= ~BAM_ENTRY_UPGFSMODULE;
8106                        BAM_DPRINTF((D_ADDING_MODULE_DOLLAR, fcn, module));
8107                }
```

```
8108         }

8110         BAM_DPRINTF((D_RETURN_RET, fcn, i));
8111         return (i);
8112 }

8114 int
8115 root_optional(char *osroot, char *menu_root)
8116 {
8117         char                    *ospecial;
8118         char                    *mspecial;
8119         char                    *slash;
8120         int                     root_opt;
8121         int                     ret1;
8122         int                     ret2;
8123         const char              *fcn = "root_optional()";

8125         BAM_DPRINTF((D_FUNC_ENTRY2, fcn, osroot, menu_root));

8127         /*
8128          * For all filesystems except ZFS, a straight compare of osroot
8129          * and menu_root will tell us if root is optional.
8130          * For ZFS, the situation is complicated by the fact that
8131          * menu_root and osroot are always different
8132          */
8133         ret1 = is_zfs(osroot);
8134         ret2 = is_zfs(menu_root);
8135         INJECT_ERROR1("ROOT_OPT_NOT_ZFS", ret1 = 0);
8136         if (!ret1 || !ret2) {
8137                 BAM_DPRINTF((D_ROOT_OPT_NOT_ZFS, fcn, osroot, menu_root));
8138                 root_opt = (strcmp(osroot, menu_root) == 0);
8139                 goto out;
8140         }

8142         ospecial = get_special(osroot);
8143         INJECT_ERROR1("ROOT_OPTIONAL_OSPECIAL", ospecial = NULL);
8144         if (ospecial == NULL) {
8145                 bam_error(GET_OSROOT_SPECIAL_ERR, osroot);
8146                 return (0);
8147         }
8148         BAM_DPRINTF((D_ROOT_OPTIONAL_OSPECIAL, fcn, ospecial, osroot));

8150         mspecial = get_special(menu_root);
8151         INJECT_ERROR1("ROOT_OPTIONAL_MSPECIAL", mspecial = NULL);
8152         if (mspecial == NULL) {
8153                 bam_error(GET_MENU_ROOT_SPECIAL_ERR, menu_root);
8154                 free(ospecial);
8155                 return (0);
8156         }
8157         BAM_DPRINTF((D_ROOT_OPTIONAL_MSPECIAL, fcn, mspecial, menu_root));

8159         slash = strchr(ospecial, '/');
8160         if (slash)
8161                 *slash = '\0';
8162         BAM_DPRINTF((D_ROOT_OPTIONAL_FIXED_OSPECIAL, fcn, ospecial, osroot));

8164         root_opt = (strcmp(ospecial, mspecial) == 0);

8166         free(ospecial);
8167         free(mspecial);

8169 out:
8170         INJECT_ERROR1("ROOT_OPTIONAL_NO", root_opt = 0);
8171         INJECT_ERROR1("ROOT_OPTIONAL_YES", root_opt = 1);
8172         if (root_opt) {
8173                 BAM_DPRINTF((D_RETURN_SUCCESS, fcn));
```



```
8174         } else {
8175                 BAM_DPRINTF((D_RETURN_FAILURE, fcn));
8176         }

8178         return (root_opt);
8179 }

8181 /*ARGSUSED*/
8182 static error_t
8183 update_entry(menu_t *mp, char *menu_root, char *osdev)
8184 {
8185         int             entry;
8186         char            *grubsign;
8187         char            *grubroot;
8188         char            *title;
8189         char            osroot[PATH_MAX];
8190         char            *failsafe_kernel = NULL;
8191         struct stat     sbuf;
8192         char            failsafe[256];
8193         char            failsafe_64[256];
8194         int             ret;
8195         const char      *fcn = "update_entry()";

8197         assert(mp);
8198         assert(menu_root);
8199         assert(osdev);
8200         assert(bam_root);

8202         BAM_DPRINTF((D_FUNC_ENTRY3, fcn, menu_root, osdev, bam_root));

8204         (void) strlcpy(osroot, bam_root, sizeof (osroot));

8206         title = get_title(osroot);
8207         assert(title);

8209         grubsign = get_grubsign(osroot, osdev);
8210         INJECT_ERROR1("GET_GRUBSIGN_FAIL", grubsign = NULL);
8211         if (grubsign == NULL) {
8212                 bam_error(GET_GRUBSIGN_ERROR, osroot, osdev);
8213                 return (BAM_ERROR);
8214         }

8216         /*
8217          * It is not a fatal error if get_grubroot() fails
8218          * We no longer rely on biosdev to populate the
8219          * menu
8220          */
8221         grubroot = get_grubroot(osroot, osdev, menu_root);
8222         INJECT_ERROR1("GET_GRUBROOT_FAIL", grubroot = NULL);
8223         if (grubroot) {
8224                 BAM_DPRINTF((D_GET_GRUBROOT_SUCCESS,
8225                     fcn, osroot, osdev, menu_root));
8226         } else {
8227                 BAM_DPRINTF((D_GET_GRUBROOT_FAILURE,
8228                     fcn, osroot, osdev, menu_root));
8229         }

8231         /* add the entry for normal Solaris */
8232         INJECT_ERROR1("UPDATE_ENTRY_MULTIBOOT",
8233             bam_direct = BAM_DIRECT_MULTIBOOT);
8234         if (bam_direct == BAM_DIRECT_DBOOT) {
8235                 entry = update_boot_entry(mp, title, grubsign, grubroot,
8236                     (bam_zfs ? DIRECT_BOOT_KERNEL_ZFS : DIRECT_BOOT_KERNEL),
8237                     NULL, DIRECT_BOOT_ARCHIVE,
8238                     root_optional(osroot, menu_root));
8239                 BAM_DPRINTF((D_UPDATED_BOOT_ENTRY, fcn, bam_zfs, grubsign));
```

```
8240                if ((entry != BAM_ERROR) && (bam_is_hv == BAM_HV_PRESENT)) {
8241                        (void) update_boot_entry(mp, NEW_HV_ENTRY, grubsign,
8242                            grubroot, XEN_MENU, bam_zfs ?
8243                            XEN_KERNEL_MODULE_LINE_ZFS : XEN_KERNEL_MODULE_LINE,
8244                            DIRECT_BOOT_ARCHIVE,
8245                            root_optional(osroot, menu_root));
8246                        BAM_DPRINTF((D_UPDATED_HV_ENTRY,
8247                            fcn, bam_zfs, grubsign));
8248                }
8249        } else {
8250                entry = update_boot_entry(mp, title, grubsign, grubroot,
8251                    MULTI_BOOT, NULL, MULTIBOOT_ARCHIVE,
8252                    root_optional(osroot, menu_root));
8253
8254                BAM_DPRINTF((D_UPDATED_MULTIBOOT_ENTRY, fcn, grubsign));
8255        }
8256
8257        /*
8258         * Add the entry for failsafe archive.  On a bfu'd system, the
8259         * failsafe may be different than the installed kernel.
8260         */
8261        (void) snprintf(failsafe, sizeof (failsafe), "%s%s",
8262            osroot, FAILSAFE_ARCHIVE_32);
8263        (void) snprintf(failsafe_64, sizeof (failsafe_64), "%s%s",
8264            osroot, FAILSAFE_ARCHIVE_64);
8265
8266        /*
8267         * Check if at least one of the two archives exists
8268         * Using $ISADIR as the default line, we have an entry which works
8269         * for both the cases.
8270         */
8271
8272        if (stat(failsafe, &sbuf) == 0 || stat(failsafe_64, &sbuf) == 0) {
8273
8274                /* Figure out where the kernel line should point */
8275                (void) snprintf(failsafe, sizeof (failsafe), "%s%s", osroot,
8276                    DIRECT_BOOT_FAILSAFE_32);
8277                (void) snprintf(failsafe_64, sizeof (failsafe_64), "%s%s",
8278                    osroot, DIRECT_BOOT_FAILSAFE_64);
8279                if (stat(failsafe, &sbuf) == 0 ||
8280                    stat(failsafe_64, &sbuf) == 0) {
8281                        failsafe_kernel = DIRECT_BOOT_FAILSAFE_LINE;
8282                } else {
8283                        (void) snprintf(failsafe, sizeof (failsafe), "%s%s",
8284                            osroot, MULTI_BOOT_FAILSAFE);
8285                        if (stat(failsafe, &sbuf) == 0) {
8286                                failsafe_kernel = MULTI_BOOT_FAILSAFE_LINE;
8287                        }
8288                }
8289                if (failsafe_kernel != NULL) {
8290                        (void) update_boot_entry(mp, FAILSAFE_TITLE, grubsign,
8291                            grubroot, failsafe_kernel, NULL, FAILSAFE_ARCHIVE,
8292                            root_optional(osroot, menu_root));
8293                        BAM_DPRINTF((D_UPDATED_FAILSAFE_ENTRY, fcn,
8294                            failsafe_kernel));
8295                }
8296        }
8297        free(grubroot);
8298
8299        INJECT_ERROR1("UPDATE_ENTRY_ERROR", entry = BAM_ERROR);
8300        if (entry == BAM_ERROR) {
8301                bam_error(FAILED_TO_ADD_BOOT_ENTRY, title, grubsign);
8302                free(grubsign);
8303                return (BAM_ERROR);
8304        }
8305        free(grubsign);
```



```
8307        update_numbering(mp);
8308        ret = set_global(mp, menu_cmds[DEFAULT_CMD], entry);
8309        INJECT_ERROR1("SET_DEFAULT_ERROR", ret = BAM_ERROR);
8310        if (ret == BAM_ERROR) {
8311                bam_error(SET_DEFAULT_FAILED, entry);
8312        }
8313        BAM_DPRINTF((D_RETURN_SUCCESS, fcn));
8314        return (BAM_WRITE);
8315 }
8316
8317 static void
8318 save_default_entry(menu_t *mp, const char *which)
8319 {
8320        int             lineNum;
8321        int             entryNum;
8322        int             entry = 0;      /* default is 0 */
8323        char            linebuf[BAM_MAXLINE];
8324        line_t          *lp = mp->curdefault;
8325        const char      *fcn = "save_default_entry()";
8326
8327        if (mp->start) {
8328                lineNum = mp->end->lineNum;
8329                entryNum = mp->end->entryNum;
8330        } else {
8331                lineNum = LINE_INIT;
8332                entryNum = ENTRY_INIT;
8333        }
8334
8335        if (lp)
8336                entry = s_strtol(lp->arg);
8337
8338        (void) snprintf(linebuf, sizeof (linebuf), "#%s%d", which, entry);
8339        BAM_DPRINTF((D_SAVING_DEFAULT_TO, fcn, linebuf));
8340        line_parser(mp, linebuf, &lineNum, &entryNum);
8341        BAM_DPRINTF((D_SAVED_DEFAULT_TO, fcn, lineNum, entryNum));
8342 }
8343
8344 static void
8345 restore_default_entry(menu_t *mp, const char *which, line_t *lp)
8346 {
8347        int             entry;
8348        char            *str;
8349        const char      *fcn = "restore_default_entry()";
8350
8351        if (lp == NULL) {
8352                BAM_DPRINTF((D_RESTORE_DEFAULT_NULL, fcn));
8353                return;         /* nothing to restore */
8354        }
8355
8356        BAM_DPRINTF((D_RESTORE_DEFAULT_STR, fcn, which));
8357
8358        str = lp->arg + strlen(which);
8359        entry = s_strtol(str);
8360        (void) set_global(mp, menu_cmds[DEFAULT_CMD], entry);
8361
8362        BAM_DPRINTF((D_RESTORED_DEFAULT_TO, fcn, entry));
8363
8364        /* delete saved old default line */
8365        unlink_line(mp, lp);
8366        line_free(lp);
8367 }
8368
8369 /*
8370  * This function is for supporting reboot with args.
8371  * The opt value can be:
```

```
 * NULL         delete temp entry, if present
 * entry=<n>    switches default entry to <n>
 * else         treated as boot-args and setup a temperary menu entry
 *              and make it the default
 * Note that we are always rebooting the current OS instance
 * so osroot == / always.
 */
#define REBOOT_TITLE    "Solaris_reboot_transient"

/*ARGSUSED*/
static error_t
update_temp(menu_t *mp, char *dummy, char *opt)
{
        int             entry;
        char            *osdev;
        char            *fstype;
        char            *sign;
        char            *opt_ptr;
        char            *path;
        char            kernbuf[BUFSIZ];
        char            args_buf[BUFSIZ];
        char            signbuf[PATH_MAX];
        int             ret;
        const char      *fcn = "update_temp()";

        assert(mp);
        assert(dummy == NULL);

        /* opt can be NULL */
        BAM_DPRINTF((D_FUNC_ENTRY1, fcn, opt ? opt : "<NULL>"));
        BAM_DPRINTF((D_BAM_ROOT, fcn, bam_alt_root, bam_root));

        if (bam_alt_root || bam_rootlen != 1 ||
            strcmp(bam_root, "/") != 0 ||
            strcmp(rootbuf, "/") != 0) {
                bam_error(ALT_ROOT_INVALID, bam_root);
                return (BAM_ERROR);
        }

        /* If no option, delete exiting reboot menu entry */
        if (opt == NULL) {
                entry_t         *ent;
                BAM_DPRINTF((D_OPT_NULL, fcn));
                ent = find_boot_entry(mp, REBOOT_TITLE, NULL, NULL,
                    NULL, NULL, 0, &entry);
                if (ent == NULL) {      /* not found is ok */
                        BAM_DPRINTF((D_TRANSIENT_NOTFOUND, fcn));
                        return (BAM_SUCCESS);
                }
                (void) delete_boot_entry(mp, entry, DBE_PRINTERR);
                restore_default_entry(mp, BAM_OLDDEF, mp->olddefault);
                mp->olddefault = NULL;
                BAM_DPRINTF((D_RESTORED_DEFAULT, fcn));
                BAM_DPRINTF((D_RETURN_SUCCESS, fcn));
                return (BAM_WRITE);
        }

        /* if entry= is specified, set the default entry */
        if (strncmp(opt, "entry=", strlen("entry=")) == 0) {
                int entryNum = s_strtol(opt + strlen("entry="));
                BAM_DPRINTF((D_ENTRY_EQUALS, fcn, opt));
                if (selector(mp, opt, &entry, NULL) == BAM_SUCCESS) {
                        /* this is entry=# option */
                        ret = set_global(mp, menu_cmds[DEFAULT_CMD], entry);
                        BAM_DPRINTF((D_ENTRY_SET_IS, fcn, entry, ret));
                        return (ret);
```



```
                } else {
                        bam_error(SET_DEFAULT_FAILED, entryNum);
                        return (BAM_ERROR);
                }
        }

        /*
         * add a new menu entry based on opt and make it the default
         */

        fstype = get_fstype("/");
        INJECT_ERROR1("REBOOT_FSTYPE_NULL", fstype = NULL);
        if (fstype == NULL) {
                bam_error(REBOOT_FSTYPE_FAILED);
                return (BAM_ERROR);
        }

        osdev = get_special("/");
        INJECT_ERROR1("REBOOT_SPECIAL_NULL", osdev = NULL);
        if (osdev == NULL) {
                free(fstype);
                bam_error(REBOOT_SPECIAL_FAILED);
                return (BAM_ERROR);
        }

        sign = find_existing_sign("/", osdev, fstype);
        sign = strchr(sign, '_') + 1;
#endif /* ! codereview */
        INJECT_ERROR1("REBOOT_SIGN_NULL", sign = NULL);
        if (sign == NULL) {
                free(fstype);
                free(osdev);
                bam_error(REBOOT_SIGN_FAILED);
                return (BAM_ERROR);
        }

        free(osdev);
        (void) strlcpy(signbuf, sign, sizeof (signbuf));
        free(sign);

        assert(strchr(signbuf, '(') == NULL && strchr(signbuf, ',') == NULL &&
            strchr(signbuf, ')') == NULL);

        /*
         * There is no alternate root while doing reboot with args
         * This version of bootadm is only delivered with a DBOOT
         * version of Solaris.
         */
        INJECT_ERROR1("REBOOT_NOT_DBOOT", bam_direct = BAM_DIRECT_MULTIBOOT);
        if (bam_direct != BAM_DIRECT_DBOOT) {
                free(fstype);
                bam_error(REBOOT_DIRECT_FAILED);
                return (BAM_ERROR);
        }

        /* add an entry for Solaris reboot */
        if (opt[0] == '-') {
                /* It's an option - first see if boot-file is set */
                ret = get_kernel(mp, KERNEL_CMD, kernbuf, sizeof (kernbuf));
                INJECT_ERROR1("REBOOT_GET_KERNEL", ret = BAM_ERROR);
                if (ret != BAM_SUCCESS) {
                        free(fstype);
                        bam_error(REBOOT_GET_KERNEL_FAILED);
                        return (BAM_ERROR);
                }
                if (kernbuf[0] == '\0')
```

```
8504                        (void) strlcpy(kernbuf, DIRECT_BOOT_KERNEL,
8505                            sizeof (kernbuf));
8506                /*
8507                 * If this is a zfs file system and kernbuf does not
8508                 * have "-B $ZFS-BOOTFS" string yet, add it.
8509                 */
8510                if (strcmp(fstype, "zfs") == 0 && !strstr(kernbuf, ZFS_BOOT)) {
8511                        (void) strlcat(kernbuf, " ", sizeof (kernbuf));
8512                        (void) strlcat(kernbuf, ZFS_BOOT, sizeof (kernbuf));
8513                }
8514                (void) strlcat(kernbuf, " ", sizeof (kernbuf));
8515                (void) strlcat(kernbuf, opt, sizeof (kernbuf));
8516                BAM_DPRINTF((D_REBOOT_OPTION, fcn, kernbuf));
8517        } else if (opt[0] == '/') {
8518                /* It's a full path, so write it out. */
8519                (void) strlcpy(kernbuf, opt, sizeof (kernbuf));

8521                /*
8522                 * If someone runs:
8523                 *
8524                 *      # eeprom boot-args='-kd'
8525                 *      # reboot /platform/i86pc/kernel/unix
8526                 *
8527                 * we want to use the boot-args as part of the boot
8528                 * line.  On the other hand, if someone runs:
8529                 *
8530                 *      # reboot "/platform/i86pc/kernel/unix -kd"
8531                 *
8532                 * we don't need to mess with boot-args.  If there's
8533                 * no space in the options string, assume we're in the
8534                 * first case.
8535                 */
8536                if (strchr(opt, ' ') == NULL) {
8537                        ret = get_kernel(mp, ARGS_CMD, args_buf,
8538                            sizeof (args_buf));
8539                        INJECT_ERROR1("REBOOT_GET_ARGS", ret = BAM_ERROR);
8540                        if (ret != BAM_SUCCESS) {
8541                                free(fstype);
8542                                bam_error(REBOOT_GET_ARGS_FAILED);
8543                                return (BAM_ERROR);
8544                        }

8546                        if (args_buf[0] != '\0') {
8547                                (void) strlcat(kernbuf, " ", sizeof (kernbuf));
8548                                (void) strlcat(kernbuf, args_buf,
8549                                    sizeof (kernbuf));
8550                        }
8551                }
8552                BAM_DPRINTF((D_REBOOT_ABSPATH, fcn, kernbuf));
8553        } else {
8554                /*
8555                 * It may be a partial path, or it may be a partial
8556                 * path followed by options.  Assume that only options
8557                 * follow a space.  If someone sends us a kernel path
8558                 * that includes a space, they deserve to be broken.
8559                 */
8560                opt_ptr = strchr(opt, ' ');
8561                if (opt_ptr != NULL) {
8562                        *opt_ptr = '\0';
8563                }

8565                path = expand_path(opt);
8566                if (path != NULL) {
8567                        (void) strlcpy(kernbuf, path, sizeof (kernbuf));
8568                        free(path);
```



```
8570                        /*
8571                         * If there were options given, use those.
8572                         * Otherwise, copy over the default options.
8573                         */
8574                        if (opt_ptr != NULL) {
8575                                /* Restore the space in opt string */
8576                                *opt_ptr = ' ';
8577                                (void) strlcat(kernbuf, opt_ptr,
8578                                    sizeof (kernbuf));
8579                        } else {
8580                                ret = get_kernel(mp, ARGS_CMD, args_buf,
8581                                    sizeof (args_buf));
8582                                INJECT_ERROR1("UPDATE_TEMP_PARTIAL_ARGS",
8583                                    ret = BAM_ERROR);
8584                                if (ret != BAM_SUCCESS) {
8585                                        free(fstype);
8586                                        bam_error(REBOOT_GET_ARGS_FAILED);
8587                                        return (BAM_ERROR);
8588                                }

8590                                if (args_buf[0] != '\0') {
8591                                        (void) strlcat(kernbuf, " ",
8592                                            sizeof (kernbuf));
8593                                        (void) strlcat(kernbuf,
8594                                            args_buf, sizeof (kernbuf));
8595                                }
8596                        }
8597                        BAM_DPRINTF((D_REBOOT_RESOLVED_PARTIAL, fcn, kernbuf));
8598                } else {
8599                        free(fstype);
8600                        bam_error(UNKNOWN_KERNEL, opt);
8601                        bam_print_stderr(UNKNOWN_KERNEL_REBOOT);
8602                        return (BAM_ERROR);
8603                }
8604        }
8605        free(fstype);
8606        entry = add_boot_entry(mp, REBOOT_TITLE, signbuf, kernbuf,
8607            NULL, NULL, NULL);
8608        INJECT_ERROR1("REBOOT_ADD_BOOT_ENTRY", entry = BAM_ERROR);
8609        if (entry == BAM_ERROR) {
8610                bam_error(REBOOT_WITH_ARGS_ADD_ENTRY_FAILED);
8611                return (BAM_ERROR);
8612        }

8614        save_default_entry(mp, BAM_OLDDEF);
8615        ret = set_global(mp, menu_cmds[DEFAULT_CMD], entry);
8616        INJECT_ERROR1("REBOOT_SET_GLOBAL", ret = BAM_ERROR);
8617        if (ret == BAM_ERROR) {
8618                bam_error(REBOOT_SET_DEFAULT_FAILED, entry);
8619        }
8620        BAM_DPRINTF((D_RETURN_SUCCESS, fcn));
8621        return (BAM_WRITE);
8622 }

8624 error_t
8625 set_global(menu_t *mp, char *globalcmd, int val)
8626 {
8627        line_t          *lp;
8628        line_t          *found;
8629        line_t          *last;
8630        char            *cp;
8631        char            *str;
8632        char            prefix[BAM_MAXLINE];
8633        size_t          len;
8634        const char      *fcn = "set_global()";
```

```
8636         assert(mp);
8637         assert(globalcmd);

8639         if (strcmp(globalcmd, menu_cmds[DEFAULT_CMD]) == 0) {
8640                 INJECT_ERROR1("SET_GLOBAL_VAL_NEG", val = -1);
8641                 INJECT_ERROR1("SET_GLOBAL_MENU_EMPTY", mp->end = NULL);
8642                 INJECT_ERROR1("SET_GLOBAL_VAL_TOO_BIG", val = 100);
8643                 if (val < 0 || mp->end == NULL || val > mp->end->entryNum) {
8644                         (void) snprintf(prefix, sizeof (prefix), "%d", val);
8645                         bam_error(INVALID_ENTRY, prefix);
8646                         return (BAM_ERROR);
8647                 }
8648         }

8650         found = last = NULL;
8651         for (lp = mp->start; lp; lp = lp->next) {
8652                 if (lp->flags != BAM_GLOBAL)
8653                         continue;

8655                 last = lp; /* track the last global found */

8657                 INJECT_ERROR1("SET_GLOBAL_NULL_CMD", lp->cmd = NULL);
8658                 if (lp->cmd == NULL) {
8659                         bam_error(NO_CMD, lp->lineNum);
8660                         continue;
8661                 }
8662                 if (strcmp(globalcmd, lp->cmd) != 0)
8663                         continue;

8665                 BAM_DPRINTF((D_FOUND_GLOBAL, fcn, globalcmd));

8667                 if (found) {
8668                         bam_error(DUP_CMD, globalcmd, lp->lineNum, bam_root);
8669                 }
8670                 found = lp;
8671         }

8673         if (found == NULL) {
8674                 lp = s_calloc(1, sizeof (line_t));
8675                 if (last == NULL) {
8676                         lp->next = mp->start;
8677                         mp->start = lp;
8678                         mp->end = (mp->end) ? mp->end : lp;
8679                 } else {
8680                         lp->next = last->next;
8681                         last->next = lp;
8682                         if (lp->next == NULL)
8683                                 mp->end = lp;
8684                 }
8685                 lp->flags = BAM_GLOBAL; /* other fields not needed for writes */
8686                 len = strlen(globalcmd) + strlen(menu_cmds[SEP_CMD]);
8687                 len += 10;      /* val < 10 digits */
8688                 lp->line = s_calloc(1, len);
8689                 (void) snprintf(lp->line, len, "%s%s%d",
8690                     globalcmd, menu_cmds[SEP_CMD], val);
8691                 BAM_DPRINTF((D_SET_GLOBAL_WROTE_NEW, fcn, lp->line));
8692                 BAM_DPRINTF((D_RETURN_SUCCESS, fcn));
8693                 return (BAM_WRITE);
8694         }

8696         /*
8697          * We are changing an existing entry. Retain any prefix whitespace,
8698          * but overwrite everything else. This preserves tabs added for
8699          * readability.
8700          */
8701         str = found->line;
```



```
8702         cp = prefix;
8703         while (*str == ' ' || *str == '\t')
8704                 *(cp++) = *(str++);
8705         *cp = '\0'; /* Terminate prefix */
8706         len = strlen(prefix) + strlen(globalcmd);
8707         len += strlen(menu_cmds[SEP_CMD]) + 10;

8709         free(found->line);
8710         found->line = s_calloc(1, len);
8711         (void) snprintf(found->line, len,
8712             "%s%s%s%d", prefix, globalcmd, menu_cmds[SEP_CMD], val);

8714         BAM_DPRINTF((D_SET_GLOBAL_REPLACED, fcn, found->line));
8715         BAM_DPRINTF((D_RETURN_SUCCESS, fcn));
8716         return (BAM_WRITE); /* need a write to menu */
8717 }

8719 /*
8720  * partial_path may be anything like "kernel/unix" or "kmdb".  Try to
8721  * expand it to a full unix path.  The calling function is expected to
8722  * output a message if an error occurs and NULL is returned.
8723  */
8724 static char *
8725 expand_path(const char *partial_path)
8726 {
8727         int             new_path_len;
8728         char            *new_path;
8729         char            new_path2[PATH_MAX];
8730         struct stat     sb;
8731         const char      *fcn = "expand_path()";

8733         new_path_len = strlen(partial_path) + 64;
8734         new_path = s_calloc(1, new_path_len);

8736         /* First, try the simplest case - something like "kernel/unix" */
8737         (void) snprintf(new_path, new_path_len, "/platform/i86pc/%s",
8738             partial_path);
8739         if (stat(new_path, &sb) == 0) {
8740                 BAM_DPRINTF((D_EXPAND_PATH, fcn, new_path));
8741                 return (new_path);
8742         }

8744         if (strcmp(partial_path, "kmdb") == 0) {
8745                 (void) snprintf(new_path, new_path_len, "%s -k",
8746                     DIRECT_BOOT_KERNEL);
8747                 BAM_DPRINTF((D_EXPAND_PATH, fcn, new_path));
8748                 return (new_path);
8749         }

8751         /*
8752          * We've quickly reached unsupported usage.  Try once more to
8753          * see if we were just given a glom name.
8754          */
8755         (void) snprintf(new_path, new_path_len, "/platform/i86pc/%s/unix",
8756             partial_path);
8757         (void) snprintf(new_path2, PATH_MAX, "/platform/i86pc/%s/amd64/unix",
8758             partial_path);
8759         if (stat(new_path, &sb) == 0) {
8760                 if (stat(new_path2, &sb) == 0) {
8761                         /*
8762                          * We matched both, so we actually
8763                          * want to write the $ISADIR version.
8764                          */
8765                         (void) snprintf(new_path, new_path_len,
8766                             "/platform/i86pc/kernel/%s/$ISADIR/unix",
8767                             partial_path);
```

```
8768                }
8769                BAM_DPRINTF((D_EXPAND_PATH, fcn, new_path));
8770                return (new_path);
8771        }

8773        free(new_path);
8774        BAM_DPRINTF((D_RETURN_FAILURE, fcn));
8775        return (NULL);
8776 }

8778 /*
8779  * The kernel cmd and arg have been changed, so
8780  * check whether the archive line needs to change.
8781  */
8782 static void
8783 set_archive_line(entry_t *entryp, line_t *kernelp)
8784 {
8785        line_t          *lp = entryp->start;
8786        char            *new_archive;
8787        menu_cmd_t      m_cmd;
8788        const char      *fcn = "set_archive_line()";

8790        for (; lp != NULL; lp = lp->next) {
8791                if (lp->cmd != NULL && strncmp(lp->cmd, menu_cmds[MODULE_CMD],
8792                    sizeof (menu_cmds[MODULE_CMD]) - 1) == 0) {
8793                        break;
8794                }

8796                INJECT_ERROR1("SET_ARCHIVE_LINE_END_ENTRY", lp = entryp->end);
8797                if (lp == entryp->end) {
8798                        BAM_DPRINTF((D_ARCHIVE_LINE_NONE, fcn,
8799                            entryp->entryNum));
8800                        return;
8801                }
8802        }
8803        INJECT_ERROR1("SET_ARCHIVE_LINE_END_MENU", lp = NULL);
8804        if (lp == NULL) {
8805                BAM_DPRINTF((D_ARCHIVE_LINE_NONE, fcn, entryp->entryNum));
8806                return;
8807        }

8809        if (strstr(kernelp->arg, "$ISADIR") != NULL) {
8810                new_archive = DIRECT_BOOT_ARCHIVE;
8811                m_cmd = MODULE_DOLLAR_CMD;
8812        } else if (strstr(kernelp->arg, "amd64") != NULL) {
8813                new_archive = DIRECT_BOOT_ARCHIVE_64;
8814                m_cmd = MODULE_CMD;
8815        } else {
8816                new_archive = DIRECT_BOOT_ARCHIVE_32;
8817                m_cmd = MODULE_CMD;
8818        }

8820        if (strcmp(lp->arg, new_archive) == 0) {
8821                BAM_DPRINTF((D_ARCHIVE_LINE_NOCHANGE, fcn, lp->arg));
8822                return;
8823        }

8825        if (lp->cmd != NULL && strcmp(lp->cmd, menu_cmds[m_cmd]) != 0) {
8826                free(lp->cmd);
8827                lp->cmd = s_strdup(menu_cmds[m_cmd]);
8828        }

8830        free(lp->arg);
8831        lp->arg = s_strdup(new_archive);
8832        update_line(lp);
8833        BAM_DPRINTF((D_ARCHIVE_LINE_REPLACED, fcn, lp->line));
```



```
8834 }

8836 /*
8837  * Title for an entry to set properties that once went in bootenv.rc.
8838  */
8839 #define BOOTENV_RC_TITLE        "Solaris bootenv rc"

8841 /*
8842  * If path is NULL, return the kernel (optnum == KERNEL_CMD) or arguments
8843  * (optnum == ARGS_CMD) in the argument buf.  If path is a zero-length
8844  * string, reset the value to the default.  If path is a non-zero-length
8845  * string, set the kernel or arguments.
8846  */
8847 static error_t
8848 get_set_kernel(
8849        menu_t *mp,
8850        menu_cmd_t optnum,
8851        char *path,
8852        char *buf,
8853        size_t bufsize)
8854 {
8855        int             entryNum;
8856        int             rv = BAM_SUCCESS;
8857        int             free_new_path = 0;
8858        entry_t         *entryp;
8859        line_t          *ptr;
8860        line_t          *kernelp;
8861        char            *new_arg;
8862        char            *old_args;
8863        char            *space;
8864        char            *new_path;
8865        char            old_space;
8866        size_t          old_kernel_len;
8867        size_t          new_str_len;
8868        char            *fstype;
8869        char            *osdev;
8870        char            *sign;
8871        char            signbuf[PATH_MAX];
8872        int             ret;
8873        const char      *fcn = "get_set_kernel()";

8875        assert(bufsize > 0);

8877        ptr = kernelp = NULL;
8878        new_arg = old_args = space = NULL;
8879        new_path = NULL;
8880        buf[0] = '\0';

8882        INJECT_ERROR1("GET_SET_KERNEL_NOT_DBOOT",
8883            bam_direct = BAM_DIRECT_MULTIBOOT);
8884        if (bam_direct != BAM_DIRECT_DBOOT) {
8885                bam_error(NOT_DBOOT, optnum == KERNEL_CMD ? "kernel" : "args");
8886                return (BAM_ERROR);
8887        }

8889        /*
8890         * If a user changed the default entry to a non-bootadm controlled
8891         * one, we don't want to mess with it.  Just print an error and
8892         * return.
8893         */
8894        if (mp->curdefault) {
8895                entryNum = s_strtol(mp->curdefault->arg);
8896                for (entryp = mp->entries; entryp; entryp = entryp->next) {
8897                        if (entryp->entryNum == entryNum)
8898                                break;
8899                }
```

```
8900                if ((entryp != NULL) &&
8901                    ((entryp->flags & (BAM_ENTRY_BOOTADM|BAM_ENTRY_LU)) == 0)) {
8902                        bam_error(DEFAULT_NOT_BAM);
8903                        return (BAM_ERROR);
8904                }
8905        }

8907        entryp = find_boot_entry(mp, BOOTENV_RC_TITLE, NULL, NULL, NULL, NULL,
8908            0, &entryNum);

8910        if (entryp != NULL) {
8911                for (ptr = entryp->start; ptr && ptr != entryp->end;
8912                    ptr = ptr->next) {
8913                        if (strncmp(ptr->cmd, menu_cmds[KERNEL_CMD],
8914                            sizeof (menu_cmds[KERNEL_CMD]) - 2) == 0) {
5380                            sizeof (menu_cmds[KERNEL_CMD]) - 1) == 0) {
8915                                kernelp = ptr;
8916                                break;
8917                        }
8918                }
8919                if (kernelp == NULL) {
8920                        bam_error(NO_KERNEL, entryNum);
8921                        return (BAM_ERROR);
8922                }

8924                old_kernel_len = strcspn(kernelp->arg, " \t");
8925                space = old_args = kernelp->arg + old_kernel_len;
8926                while ((*old_args == ' ') || (*old_args == '\t'))
8927                        old_args++;
8928        }

8930        if (path == NULL) {
8931                if (entryp == NULL) {
8932                        BAM_DPRINTF((D_GET_SET_KERNEL_NO_RC, fcn));
8933                        BAM_DPRINTF((D_RETURN_SUCCESS, fcn));
8934                        return (BAM_SUCCESS);
8935                }
8936                assert(kernelp);
8937                if (optnum == ARGS_CMD) {
8938                        if (old_args[0] != '\0') {
8939                                (void) strlcpy(buf, old_args, bufsize);
8940                                BAM_DPRINTF((D_GET_SET_KERNEL_ARGS, fcn, buf));
8941                        }
8942                } else {
8943                        /*
8944                         * We need to print the kernel, so we just turn the
8945                         * first space into a '\0' and print the beginning.
8946                         * We don't print anything if it's the default kernel.
8947                         */
8948                        old_space = *space;
8949                        *space = '\0';
8950                        if (strcmp(kernelp->arg, DIRECT_BOOT_KERNEL) != 0) {
8951                                (void) strlcpy(buf, kernelp->arg, bufsize);
8952                                BAM_DPRINTF((D_GET_SET_KERNEL_KERN, fcn, buf));
8953                        }
8954                        *space = old_space;
8955                }
8956                BAM_DPRINTF((D_RETURN_SUCCESS, fcn));
8957                return (BAM_SUCCESS);
8958        }

8960        /*
8961         * First, check if we're resetting an entry to the default.
8962         */
8963        if ((path[0] == '\0') ||
8964            ((optnum == KERNEL_CMD) &&
```



```
8965            (strcmp(path, DIRECT_BOOT_KERNEL) == 0))) {
8966                if ((entryp == NULL) || (kernelp == NULL)) {
8967                        /* No previous entry, it's already the default */
8968                        BAM_DPRINTF((D_GET_SET_KERNEL_ALREADY, fcn));
8969                        return (BAM_SUCCESS);
8970                }

8972                /*
8973                 * Check if we can delete the entry.  If we're resetting the
8974                 * kernel command, and the args is already empty, or if we're
8975                 * resetting the args command, and the kernel is already the
8976                 * default, we can restore the old default and delete the entry.
8977                 */
8978                if (((optnum == KERNEL_CMD) &&
8979                    ((old_args == NULL) || (old_args[0] == '\0'))) ||
8980                    ((optnum == ARGS_CMD) &&
8981                    (strncmp(kernelp->arg, DIRECT_BOOT_KERNEL,
8982                    sizeof (DIRECT_BOOT_KERNEL) - 1) == 0))) {
8983                        kernelp = NULL;
8984                        (void) delete_boot_entry(mp, entryNum, DBE_PRINTERR);
8985                        restore_default_entry(mp, BAM_OLD_RC_DEF,
8986                            mp->old_rc_default);
8987                        mp->old_rc_default = NULL;
8988                        rv = BAM_WRITE;
8989                        BAM_DPRINTF((D_GET_SET_KERNEL_RESTORE_DEFAULT, fcn));
8990                        goto done;
8991                }

8993                if (optnum == KERNEL_CMD) {
8994                        /*
8995                         * At this point, we've already checked that old_args
8996                         * and entryp are valid pointers.  The "+ 2" is for
8997                         * a space a the string termination character.
8998                         */
8999                        new_str_len = (sizeof (DIRECT_BOOT_KERNEL) - 1) +
9000                            strlen(old_args) + 2;
9001                        new_arg = s_calloc(1, new_str_len);
9002                        (void) snprintf(new_arg, new_str_len, "%s",
9003                            DIRECT_BOOT_KERNEL);
5468                        (void) snprintf(new_arg, new_str_len, "%s %s",
5469                            DIRECT_BOOT_KERNEL, old_args);
9004                        free(kernelp->arg);
9005                        kernelp->arg = new_arg;

9007                        /*
9008                         * We have changed the kernel line, so we may need
9009                         * to update the archive line as well.
9010                         */
9011                        set_archive_line(entryp, kernelp);
9012                        BAM_DPRINTF((D_GET_SET_KERNEL_RESET_KERNEL_SET_ARG,
9013                            fcn, kernelp->arg));
9014                } else {
9015                        /*
9016                         * We're resetting the boot args to nothing, so
9017                         * we only need to copy the kernel.  We've already
9018                         * checked that the kernel is not the default.
9019                         */
9020                        new_arg = s_calloc(1, old_kernel_len + 1);
9021                        (void) snprintf(new_arg, old_kernel_len + 1, "%s",
9022                            kernelp->arg);
9023                        free(kernelp->arg);
9024                        kernelp->arg = new_arg;
9025                        BAM_DPRINTF((D_GET_SET_KERNEL_RESET_ARG_SET_KERNEL,
9026                            fcn, kernelp->arg));
9027                }
9028                rv = BAM_WRITE;
```

```
9029                    goto done;
9030            }

9032            /*
9033             * Expand the kernel file to a full path, if necessary
9034             */
9035            if ((optnum == KERNEL_CMD) && (path[0] != '/')) {
9036                    new_path = expand_path(path);
9037                    if (new_path == NULL) {
9038                            bam_error(UNKNOWN_KERNEL, path);
9039                            BAM_DPRINTF((D_RETURN_FAILURE, fcn));
9040                            return (BAM_ERROR);
9041                    }
9042                    free_new_path = 1;
9043            } else {
9044                    new_path = path;
9045                    free_new_path = 0;
9046            }

9048            /*
9049             * At this point, we know we're setting a new value.  First, take care
9050             * of the case where there was no previous entry.
9051             */
9052            if (entryp == NULL) {

9054                    /* Similar to code in update_temp */
9055                    fstype = get_fstype("/");
9056                    INJECT_ERROR1("GET_SET_KERNEL_FSTYPE", fstype = NULL);
9057                    if (fstype == NULL) {
9058                            bam_error(BOOTENV_FSTYPE_FAILED);
9059                            rv = BAM_ERROR;
9060                            goto done;
9061                    }

9063                    osdev = get_special("/");
9064                    INJECT_ERROR1("GET_SET_KERNEL_SPECIAL", osdev = NULL);
9065                    if (osdev == NULL) {
9066                            free(fstype);
9067                            bam_error(BOOTENV_SPECIAL_FAILED);
9068                            rv = BAM_ERROR;
9069                            goto done;
9070                    }

9072                    sign = find_existing_sign("/", osdev, fstype);
9073                    INJECT_ERROR1("GET_SET_KERNEL_SIGN", sign = NULL);
9074                    if (sign == NULL) {
9075                            free(fstype);
9076                            free(osdev);
9077                            bam_error(BOOTENV_SIGN_FAILED);
9078                            rv = BAM_ERROR;
9079                            goto done;
9080                    }

9082                    free(osdev);
9083                    (void) strlcpy(signbuf, sign, sizeof (signbuf));
9084                    free(sign);
9085                    assert(strchr(signbuf, '(') == NULL &&
9086                        strchr(signbuf, ',') == NULL &&
9087                        strchr(signbuf, ')') == NULL);

9089                    if (optnum == KERNEL_CMD) {
9090                            if (strcmp(fstype, "zfs") == 0) {
9091                                    new_str_len = strlen(new_path) +
9092                                        strlen(ZFS_BOOT) + 8;
9093                                    new_arg = s_calloc(1, new_str_len);
9094                                    (void) snprintf(new_arg, new_str_len, "%s %s",
```



```
9095                                        new_path, ZFS_BOOT);
9096                                    BAM_DPRINTF((D_GET_SET_KERNEL_NEW_KERN, fcn,
9097                                        new_arg));
9098                                    entryNum = add_boot_entry(mp, BOOTENV_RC_TITLE,
9099                                        signbuf, new_arg, NULL, NULL, NULL);
9100                                    free(new_arg);
9101                            } else {
9102                                    BAM_DPRINTF((D_GET_SET_KERNEL_NEW_KERN, fcn,
9103                                        new_path));
9104                                    entryNum = add_boot_entry(mp, BOOTENV_RC_TITLE,
9105                                        signbuf, new_path, NULL, NULL, NULL);
9106                            }
9107                    } else {
9108                            new_str_len = strlen(path) + 8;
9109                            if (strcmp(fstype, "zfs") == 0) {
9110                                    new_str_len += strlen(DIRECT_BOOT_KERNEL_ZFS);
9111                                    new_arg = s_calloc(1, new_str_len);
9112                                    (void) snprintf(new_arg, new_str_len, "%s %s",
9113                                        DIRECT_BOOT_KERNEL_ZFS, path);
9114                            } else {
9115                                    new_str_len += strlen(DIRECT_BOOT_KERNEL);
9116                                    new_arg = s_calloc(1, new_str_len);
9117                                    (void) snprintf(new_arg, new_str_len, "%s %s",
9118                                        DIRECT_BOOT_KERNEL, path);
9119                            }

9121                            BAM_DPRINTF((D_GET_SET_KERNEL_NEW_ARG, fcn, new_arg));
9122                            entryNum = add_boot_entry(mp, BOOTENV_RC_TITLE,
9123                                signbuf, new_arg, NULL, DIRECT_BOOT_ARCHIVE, NULL);
9124                            free(new_arg);
9125                    }
9126                    free(fstype);
9127                    INJECT_ERROR1("GET_SET_KERNEL_ADD_BOOT_ENTRY",
9128                        entryNum = BAM_ERROR);
9129                    if (entryNum == BAM_ERROR) {
9130                            bam_error(GET_SET_KERNEL_ADD_BOOT_ENTRY,
9131                                BOOTENV_RC_TITLE);
9132                            rv = BAM_ERROR;
9133                            goto done;
9134                    }
9135                    save_default_entry(mp, BAM_OLD_RC_DEF);
9136                    ret = set_global(mp, menu_cmds[DEFAULT_CMD], entryNum);
9137                    INJECT_ERROR1("GET_SET_KERNEL_SET_GLOBAL", ret = BAM_ERROR);
9138                    if (ret == BAM_ERROR) {
9139                            bam_error(GET_SET_KERNEL_SET_GLOBAL, entryNum);
9140                    }
9141                    rv = BAM_WRITE;
9142                    goto done;
9143            }

9145            /*
9146             * There was already an bootenv entry which we need to edit.
9147             */
9148            if (optnum == KERNEL_CMD) {
9149                    new_str_len = strlen(new_path) + strlen(old_args) + 2;
9150                    new_arg = s_calloc(1, new_str_len);
9151                    (void) snprintf(new_arg, new_str_len, "%s %s", new_path,
9152                        old_args);
9153                    free(kernelp->arg);
9154                    kernelp->arg = new_arg;

9156                    /*
9157                     * If we have changed the kernel line, we may need to update
9158                     * the archive line as well.
9159                     */
9160                    set_archive_line(entryp, kernelp);
```

```
9161                 BAM_DPRINTF((D_GET_SET_KERNEL_REPLACED_KERNEL_SAME_ARG, fcn,
9162                     kernelp->arg));
9163         } else {
9164                 kernelp = kernelp->next;
9165                 new_str_len = strlen(kernelp->arg) + strlen(path) + 8;
5630                 new_str_len = old_kernel_len + strlen(path) + 8;
9166                 new_arg = s_calloc(1, new_str_len);
9167                 (void) strncpy(new_arg, kernelp->arg, strlen(kernelp->arg));
5632                 (void) strncpy(new_arg, kernelp->arg, old_kernel_len);
9168                 (void) strlcat(new_arg, " ", new_str_len);
9169                 (void) strlcat(new_arg, path, new_str_len);
9170                 free(kernelp->arg);
9171                 kernelp->arg = new_arg;
9172                 BAM_DPRINTF((D_GET_SET_KERNEL_SAME_KERNEL_REPLACED_ARG, fcn,
9173                     kernelp->arg));
9174         }
9175         rv = BAM_WRITE;

9177 done:
9178         if ((rv == BAM_WRITE) && kernelp)
9179                 update_line(kernelp);
9180         if (free_new_path)
9181                 free(new_path);
9182         if (rv == BAM_WRITE) {
9183                 BAM_DPRINTF((D_RETURN_SUCCESS, fcn));
9184         } else {
9185                 BAM_DPRINTF((D_RETURN_FAILURE, fcn));
9186         }
9187         return (rv);
9188 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
    9360 Fri Aug 31 05:08:47 2012
new/bootadm/bootadm.h
botadm patch
**********************************************************
______unchanged_portion_omitted_

 108 /*
 109  * Menu related
 110  * menu_cmd_t and menu_cmds must be kept in sync
 111  *
 112  * The *_DOLLAR_CMD values must be 1 greater than the
 113  * respective [KERNEL|MODULE]_CMD values.
 114  */
 115 typedef enum {
 116         DEFAULT_CMD = 0,
 117         TIMEOUT_CMD,
 118         TITLE_CMD,
 119         ROOT_CMD,
 120         KERNEL_CMD,
 121         KERNEL_DOLLAR_CMD,      /* Must be KERNEL_CMD + 1 */
 122         MODULE_CMD,
 123         MODULE_DOLLAR_CMD,      /* Must be MODULE_CMD + 1 */
 124         SEP_CMD,
 125         COMMENT_CMD,
 126         CHAINLOADER_CMD,
 127         ARGS_CMD,
 128         FINDROOT_CMD,
 129         BOOTFS_CMD,
 130         KERNEL_OPTIONS_CMD,
 129         BOOTFS_CMD
 131 } menu_cmd_t;
______unchanged_portion_omitted_

 156 extern int bam_verbose;
 157 extern int bam_force;
 158 extern direct_or_multi_t bam_direct;
 159 extern hv_t bam_is_hv;
 160 extern findroot_t bam_is_findroot;
 161 extern int bam_debug;

 163 extern void bam_add_line(menu_t *mp, entry_t *entry, line_t *prev, line_t *lp);
 164 extern void update_numbering(menu_t *mp);
 165 extern error_t set_global(menu_t *, char *, int);
 166 extern error_t upgrade_menu(menu_t *, char *, char *);
 167 extern error_t cvt_to_hyper(menu_t *, char *, char *);
 168 extern error_t cvt_to_metal(menu_t *, char *, char *);
 169 extern void *s_calloc(size_t, size_t);
 170 extern void *s_realloc(void *, size_t);
 171 extern char *s_fgets(char *buf, int n, FILE *fp);
 172 extern void bam_error(char *format, ...);
 173 extern void bam_exit(int);
 174 extern void bam_print(char *, ...);
 175 extern void bam_print_stderr(char *format, ...);
 176 extern void bam_derror(char *format, ...);
 177 extern error_t get_boot_cap(const char *osroot);
 178 extern char *get_special(char *);
 179 extern char *os_to_grubdisk(char *, int);
 180 extern void update_line(line_t *);
 181 extern int add_boot_entry(menu_t *, char *, char *, char *, char *, char *,
 182     char *);
 183 extern error_t delete_boot_entry(menu_t *, int, int);
 184 extern int is_grub(const char *);
 185 extern char *get_grubsign(char *osroot, char *osdev);
 186 extern char *get_grubroot(char *osroot, char *osdev, char *menu_root);
 187 extern int root_optional(char *osroot, char *menu_root);
```



```
 188 extern void unlink_line(menu_t *mp, line_t *lp);
 189 extern void line_free(line_t *lp);
 190 extern char *s_strdup(char *);
 191 extern int is_sparc(void);

 193 #define BAM_MAXLINE     8192

 195 /* menu.lst comments created by bootadm */
 196 #define BAM_BOOTADM_HDR "---------- ADDED BY BOOTADM - DO NOT EDIT ----------"
 197 #define BAM_BOOTADM_FTR "---------------------END BOOTADM--------------------"

 199 /*
 200  * menu.lst comments create by Live Upgrade.  Note that these are the end of
 201  * the comment strings - there will be other text before them.
 202  */
 203 #define BAM_LU_HDR      " - ADDED BY LIVE UPGRADE - DO NOT EDIT  -----"
 204 #define BAM_LU_FTR      " -------------- END LIVE UPGRADE ------------"

 206 #define BAM_OLDDEF      "BOOTADM SAVED DEFAULT: "
 207 #define BAM_OLD_RC_DEF  "BOOTADM RC SAVED DEFAULT: "

 209 /*
 210  * menu.lst comment created by libbe
 211  */
 212 #define BAM_LIBBE_FTR   "============ End of LIBBE entry ============="

 214 /* Title used for failsafe entries */
 215 #define FAILSAFE_TITLE  "Solaris failsafe"

 217 /* Title used for hv entries */
 218 #define NEW_HV_ENTRY    "Solaris xVM"

 220 /* ZFS boot option */
 221 #define ZFS_BOOT        "-B $ZFS_BOOTFS"
 220 #define ZFS_BOOT        "-B $ZFS-BOOTFS"

 223 /* multiboot */
 224 #define MULTI_BOOT      "/platform/i86pc/multiboot"
 225 #define MULTI_BOOT_FAILSAFE     "/boot/multiboot"
 226 #define MULTI_BOOT_FAILSAFE_UNIX        "kernel/unix"
 227 #define MULTI_BOOT_FAILSAFE_LINE        "/boot/multiboot kernel/unix -s"

 229 /* directboot kernels */
 230 #define DIRECT_BOOT_32  "/platform/i86pc/kernel/unix"
 231 #define DIRECT_BOOT_64  "/platform/i86pc/kernel/amd64/unix"
 232 #define DIRECT_BOOT_KERNEL      "/platform/i86pc/kernel/$ISADIR/unix"
 233 #define DIRECT_BOOT_FAILSAFE_32 "/boot/platform/i86pc/kernel/unix"
 234 #define DIRECT_BOOT_FAILSAFE_64 "/boot/platform/i86pc/kernel/amd64/unix"
 235 #define DIRECT_BOOT_FAILSAFE_KERNEL \
 236         "/boot/platform/i86pc/kernel/$ISADIR/unix"
 237 #define DIRECT_BOOT_FAILSAFE_LINE       DIRECT_BOOT_FAILSAFE_KERNEL " -s"
 238 #define DIRECT_BOOT_KERNEL_ZFS  DIRECT_BOOT_KERNEL " " ZFS_BOOT
 239 #define DIRECT_BOOT_PREFIX      "/platform/i86pc/"
 240 #define KERNEL_PREFIX   "/platform/i86pc/"
 241 #define AMD_UNIX_SPACE  "/amd64/unix "
 242 #define UNIX_SPACE      "/unix "

 244 /* xVM kernels */
 245 #define XEN_KERNEL_SUBSTR "xen.gz"

 247 /* Boot archives */
 248 #define ARCHIVE_PREFIX          "/platform/"
 249 #define ARCHIVE_SUFFIX          "/boot_archive"
 250 #define CACHEDIR_SUFFIX         "/archive_cache"
 251 #define UPDATEDIR_SUFFIX        "/updates"
 252 #define DIRECT_BOOT_ARCHIVE     "/platform/i86pc/$ISADIR/boot_archive"
```

```
 253 #define DIRECT_BOOT_ARCHIVE_32  "/platform/i86pc/boot_archive"
 254 #define DIRECT_BOOT_ARCHIVE_64  "/platform/i86pc/amd64/boot_archive"
 255 #define MULTIBOOT_ARCHIVE       DIRECT_BOOT_ARCHIVE_32
 256 #define FAILSAFE_ARCHIVE        "/boot/$ISADIR/x86.miniroot-safe"
 257 #define FAILSAFE_ARCHIVE_32     "/boot/x86.miniroot-safe"
 258 #define FAILSAFE_ARCHIVE_64     "/boot/amd64/x86.miniroot-safe"
 259 #define CACHEDIR_32             "/platform/i86pc/archive_cache"
 260 #define CACHEDIR_64             "/platform/i86pc/amd64/archive_cache"
 261 #define UPDATEDIR_32            "/platform/i86pc/updates"
 262 #define UPDATEDIR_64            "/platform/i86pc/amd64/updates"

 264 /* Hypervisors */
 265 #define XEN_64                  "/boot/amd64/xen.gz"
 266 #define XEN_MENU                "/boot/$ISADIR/xen.gz"
 267 #define HYPERVISOR_KERNEL       "/platform/i86xpv/kernel/$ISADIR/unix"
 268 #define XEN_KERNEL_MODULE_LINE  HYPERVISOR_KERNEL " " HYPERVISOR_KERNEL
 269 #define XEN_KERNEL_MODULE_LINE_ZFS      \
 270         HYPERVISOR_KERNEL " " HYPERVISOR_KERNEL " " ZFS_BOOT

 272 /* Helpers */
 273 #define MKISOFS_PATH            "/usr/bin/mkisofs"
 274 #define DD_PATH_USR             "/usr/bin/dd"
 275 #define LOCKFS_PATH             "/usr/sbin/lockfs"

 277 /* A first guess at the number of entries in a menu */
 278 #define BAM_ENTRY_NUM           10

 280 /* toggle for whether delete_boot_entry prints an error message or not */
 281 #define DBE_PRINTERR            0
 282 #define DBE_QUIET               1

 284 /*
 285  * Debugging defines
 286  */
 287 #define INJECT_ERROR1(x, y)     \
 288 { \
 289         if (bam_debug) { \
 290                 char *inj = getenv("_BOOTADM_INJECT"); \
 291                 if (inj && strcmp(inj, (x)) == 0) {  \
 292                         y;      \
 293                 } \
 294         } \
 295 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   29758 Fri Aug 31 05:08:48 2012
new/bootadm/bootadm_hyper.c
bug fix
adding functionality and fixing bugs
adding functions to menuadm
menuadm to replace bootadm menu interaction
hypervisor + bug fix
**********************************************************
```

```
   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  */

  22 /*
  23  * Copyright (c) 2009, 2010, Oracle and/or its affiliates. All rights reserved.
  24  */

  26 /*
  27  * Copyright 2012 Daniil Lunev. All rights reserved.
  28  */

  30 #endif /* ! codereview */
  31 #include <stdio.h>
  32 #include <errno.h>
  33 #include <stdlib.h>
  34 #include <string.h>
  35 #include <unistd.h>
  36 #include <alloca.h>
  37 #include <ctype.h>
  38 #include <sys/types.h>

  40 #include "message.h"
  41 #include "bootadm.h"

  43 #define HYPER_KERNEL_DIR                "/platform/i86xpv/kernel"
  44 #define METAL_KERNEL_DIR                "/platform/i86pc/kernel"

  46 #define BOOTRC_FILE                     "/boot/solaris/bootenv.rc"
  47 #define ZFS_BOOTSTR                     "$ZFS_BOOTFS"
  26 #define ZFS_BOOTSTR                     "$ZFS-BOOTFS"

  49 #define BFLAG                           "-B"
  50 #define DEFAULT_SERIAL                  "9600,8,n,1"

  52 #define TTYXMODE_TO_COMNUM(ttyxmode)    ((int)(*((ttyxmode) + 3) - '''))
  53 #define COMNAME_TO_COMNUM(comname)      ((int)(*((comname) + 3) - '0'))

  55 #define WHITESPC(x)                     (x)
```



```
  57 static char *serial_config[2] = { NULL, NULL };
  58 static char *console_dev = NULL;

  60 static char *bootenv_rc_serial[2] = { NULL, NULL };
  61 static char *bootenv_rc_console = NULL;

  63 static unsigned zfs_boot = 0;

  65 /*
  66  * Append the string pointed to by "str" to the string pointed to by "orig"
  67  * adding the delimeter "delim" in between.
  68  *
  69  * Return a pointer to the new string or NULL, if we were passed a bad string.
  70  */
  71 static char *
  72 append_str(char *orig, char *str, char *delim)
  73 {
  74         char *newstr;
  75         int len;

  77         if ((str == NULL) || (delim == NULL))
  78                 return (NULL);

  80         if ((orig == NULL) || (*orig == NULL)) {
  81                 /*
  82                  * Return a pointer to a copy of the path so a caller can
  83                  * always rely upon being able to free() a returned pointer.
  84                  */
  85                 return (s_strdup(str));
  86         }

  88         len = strlen(orig) + strlen(str) + strlen(delim) + 1;
  89         if ((newstr = malloc(len)) == NULL) {
  90                 bam_error(NO_MEM, len);
  91                 bam_exit(1);
  92         }

  94         (void) snprintf(newstr, len, "%s%s%s", orig, delim, str);
  95         return (newstr);
  96 }
_____unchanged_portion_omitted_

 755 error_t
 756 cvt_to_hyper(menu_t *mp, char *osroot, char *extra_args)
 757 {
 758         const char *fcn = "cvt_to_hyper()";

 760         line_t *lp;
 761         entry_t *ent;
 762         size_t len, zfslen;

 764         char *newstr;
 765         char *osdev;

 767         char *title = NULL;
 768         char *findroot = NULL;
 769         char *bootfs = NULL;
 770         char *kernel = NULL;
 771         char *opts = NULL;
 772 #endif /* ! codereview */
 773         char *mod_kernel = NULL;
 774         char *module = NULL;
 775         char *tmp = NULL;
 776 #endif /* ! codereview */

 778         char *kern_path = NULL;
```

```
779         char *kern_bargs = NULL;

781         int curdef, newdef;
782         int kp_allocated = 0;
783         int ret = BAM_ERROR;

785         assert(osroot);

787         BAM_DPRINTF((D_FUNC_ENTRY2, fcn, osroot, extra_args));

789         /*
790          * First just check to verify osroot is a sane directory.
791          */
792         if ((osdev = get_special(osroot)) == NULL) {
793                 bam_error(CANT_FIND_SPECIAL, osroot);
794                 return (BAM_ERROR);
795         }

797         free(osdev);

799         /*
800          * While the effect is purely cosmetic, if osroot is "/" don't
801          * bother prepending it to any paths as they are constructed to
802          * begin with "/" anyway.
803          */
804         if (strcmp(osroot, "/") == 0)
805                 osroot = "";

807         /*
808          * Found the GRUB signature on the target partitions, so now get the
809          * default GRUB boot entry number from the menu.lst file
810          */
811         curdef = atoi(mp->curdefault->arg);

813         /* look for the first line of the matching boot entry */
814         for (ent = mp->entries; ((ent != NULL) && (ent->entryNum != curdef));
815             ent = ent->next)
816                 ;

818         /* couldn't find it, so error out */
819         if (ent == NULL) {
820                 bam_error(CANT_FIND_DEFAULT, curdef);
821                 goto abort;
822         }

824         /*
825          * We found the proper menu entry, so first we need to process the
826          * bootenv.rc file to look for boot options the hypervisor might need
827          * passed as kernel start options such as the console device and serial
828          * port parameters.
829          *
830          * If there's no bootenv.rc, it's not an issue.
831          */
832         parse_bootenvrc(osroot);

834         if (bootenv_rc_console != NULL)
835                 console_metal_to_hyper(bootenv_rc_console);

837         if (bootenv_rc_serial[0] != NULL)
838                 (void) serial_metal_to_hyper("ttya-mode", bootenv_rc_serial[0]);

840         if (bootenv_rc_serial[1] != NULL)
841                 (void) serial_metal_to_hyper("ttyb-mode", bootenv_rc_serial[1]);

843         /*
844          * Now process the entry itself.
```



```
845          */
846         for (lp = ent->start; lp != NULL; lp = lp->next) {
847                 /*
848                  * Process important lines from menu.lst boot entry.
849                  */
850                 if (lp->flags == BAM_TITLE) {
851                         title = strdupa(lp->arg);
852                 } else if (lp->cmd != NULL) {
853                         if (strcmp(lp->cmd, "pool_label") == 0) {
750                         if (strcmp(lp->cmd, "findroot") == 0) {
854                                 findroot = strdupa(lp->arg);
855                         } else if (strcmp(lp->cmd, "data_set") == 0) {
752                         } else if (strcmp(lp->cmd, "bootfs") == 0) {
856                                 bootfs = strdupa(lp->arg);
857                         } else if (strcmp(lp->cmd, "kernel_options") == 0) {
858                                 opts = strdupa(lp->arg);
859 #endif /* ! codereview */
860                         } else if (strcmp(lp->cmd,
861                             menu_cmds[MODULE_DOLLAR_CMD]) == 0) {
862                                 module = strdupa(lp->arg);
863                         } else if ((strcmp(lp->cmd,
864                             menu_cmds[KERNEL_DOLLAR_CMD]) == 0) &&
865                             (ret = cvt_metal_kernel(lp->arg,
866                             &kern_path)) != 0) {
867                                 if (ret < 0) {
868                                         ret = BAM_ERROR;
869                                         bam_error(KERNEL_NOT_PARSEABLE, curdef);
870                                 } else
871                                         ret = BAM_NOCHANGE;

873                                 goto abort;
874                         }
875                 }

877                 if (lp == ent->end)
878                         break;
879         }

881
882 #endif /* ! codereview */
883         /*
884          * If findroot, module or kern_path are NULL, the boot entry is
885          * malformed.
886          */
887         if (findroot == NULL) {
888                 bam_error(FINDROOT_NOT_FOUND, curdef);
889                 goto abort;
890         }

892         if (module == NULL) {
893                 bam_error(MODULE_NOT_PARSEABLE, curdef);
894                 goto abort;
895         }

897         if (kern_path == NULL) {
898                 bam_error(KERNEL_NOT_FOUND, curdef);
899                 goto abort;
900         }

902         /* assemble new kernel and module arguments from parsed values */
903         if (console_dev != NULL) {
904                 kern_bargs = s_strdup(console_dev);

906                 if (serial_config[0] != NULL) {
907                         newstr = append_str(kern_bargs, serial_config[0], " ");
908                         free(kern_bargs);
```

```
909                         kern_bargs = newstr;
910                 }
911
912                 if (serial_config[1] != NULL) {
913                         newstr = append_str(kern_bargs, serial_config[1], " ");
914                         free(kern_bargs);
915                         kern_bargs = newstr;
916                 }
917         }
918
919         if ((extra_args != NULL) && (*extra_args != NULL)) {
920                 newstr = append_str(kern_bargs, extra_args, " ");
921                 free(kern_bargs);
922                 kern_bargs = newstr;
923         }
924
925         len = strlen(osroot) + strlen(XEN_MENU) + strlen(kern_bargs) +
926             WHITESPC(1) + 1;
927
928         kernel = alloca(len);
929
930         if (kern_bargs != NULL) {
931                 if (*kern_bargs != NULL)
932                         (void) snprintf(kernel, len, "%s%s %s", osroot,
933                             XEN_MENU, kern_bargs);
934
935                 free(kern_bargs);
936         } else {
937                 (void) snprintf(kernel, len, "%s%s", osroot, XEN_MENU);
938         }
939
940         /*
941          * Change the kernel directory from the metal version to that needed for
942          * the hypervisor.  Convert either "direct boot" path to the default
943          * path.
944          */
945         if ((strcmp(kern_path, DIRECT_BOOT_32) == 0) ||
946             (strcmp(kern_path, DIRECT_BOOT_64) == 0)) {
947                 kern_path = HYPERVISOR_KERNEL;
948         } else {
949                 newstr = modify_path(kern_path, METAL_KERNEL_DIR,
950                     HYPER_KERNEL_DIR);
951                 free(kern_path);
952                 kern_path = newstr;
953                 kp_allocated = 1;
954         }
955
956         /*
957          * We need to allocate space for the kernel path (twice) plus an
958          * intervening space, possibly the ZFS boot string, and NULL,
959          * of course.
960          */
961         len = (strlen(kern_path) * 2) + WHITESPC(1) + 1;
962         zfslen = (zfs_boot ? (WHITESPC(1) + strlen(ZFS_BOOT)) : 0);
963
964         mod_kernel = alloca(len + zfslen);
965         if (opts)
966                 (void) snprintf(mod_kernel, len + strlen(opts) + 1, "%s %s %s",
967         else
968 #endif /* ! codereview */
969                 (void) snprintf(mod_kernel, len, "%s %s", kern_path, kern_path);
970
971         if (kp_allocated)
972                 free(kern_path);
973
974         if (zfs_boot) {
```



```
975                 char *zfsstr = alloca(zfslen + 1);
976
977                 (void) snprintf(zfsstr, zfslen + 1, " %s", ZFS_BOOT);
978                 (void) strcat(mod_kernel, zfsstr);
979         }
980
981         /* shut off warning messages from the entry line parser */
982         if (ent->flags & BAM_ENTRY_BOOTADM)
983                 ent->flags &= ~BAM_ENTRY_BOOTADM;
984
985         BAM_DPRINTF((D_CVT_CMD_KERN_DOLLAR, fcn, kernel));
986         BAM_DPRINTF((D_CVT_CMD_MOD_DOLLAR, fcn, mod_kernel));
987
988         if ((newdef = add_boot_entry(mp, title, findroot, kernel, mod_kernel,
989             module, bootfs)) == BAM_ERROR)
990                 return (newdef);
991
992         /*
993          * Now try to delete the current default entry from the menu and add
994          * the new hypervisor entry with the parameters we've setup.
995          */
996         if (delete_boot_entry(mp, curdef, DBE_QUIET) == BAM_SUCCESS)
997                 newdef--;
998         else
999                 bam_print(NEW_BOOT_ENTRY, title);
1000
1001         /*
1002          * If we successfully created the new entry, set the default boot
1003          * entry to that entry and let the caller know the new menu should
1004          * be written out.
1005          */
1006         return (set_global(mp, menu_cmds[DEFAULT_CMD], newdef));
1007
1008 abort:
1009         if (ret != BAM_NOCHANGE)
1010                 bam_error(HYPER_ABORT, ((*osroot == NULL) ? "/" : osroot));
1011
1012         return (ret);
1013 }
1014
1015 /*ARGSUSED*/
1016 error_t
1017 cvt_to_metal(menu_t *mp, char *osroot, char *menu_root)
1018 {
1019         const char *fcn = "cvt_to_metal()";
1020
1021         line_t *lp;
1022         entry_t *ent;
1023         size_t len, zfslen;
1024
1025         char *delim = ",";
1026         char *newstr;
1027         char *osdev;
1028
1029         char *title = NULL;
1030         char *findroot = NULL;
1031         char *bootfs = NULL;
1032         char *kernel = NULL;
1033         char *module = NULL;
1034
1035         char *barchive_path = DIRECT_BOOT_ARCHIVE;
1036         char *kern_path = NULL;
1037
1038         int curdef, newdef;
1039         int emit_bflag = 1;
1040         int ret = BAM_ERROR;
```

```
1042         assert(osroot);

1044         BAM_DPRINTF((D_FUNC_ENTRY2, fcn, osroot, ""));

1046         /*
1047          * First just check to verify osroot is a sane directory.
1048          */
1049         if ((osdev = get_special(osroot)) == NULL) {
1050                 bam_error(CANT_FIND_SPECIAL, osroot);
1051                 return (BAM_ERROR);
1052         }

1054         free(osdev);

1056         /*
1057          * Found the GRUB signature on the target partitions, so now get the
1058          * default GRUB boot entry number from the menu.lst file
1059          */
1060         curdef = atoi(mp->curdefault->arg);

1062         /* look for the first line of the matching boot entry */
1063         for (ent = mp->entries; ((ent != NULL) && (ent->entryNum != curdef));
1064             ent = ent->next)
1065                 ;

1067         /* couldn't find it, so error out */
1068         if (ent == NULL) {
1069                 bam_error(CANT_FIND_DEFAULT, curdef);
1070                 goto abort;
1071         }

1073         /*
1074          * Now process the entry itself.
1075          */
1076         for (lp = ent->start; lp != NULL; lp = lp->next) {
1077                 /*
1078                  * Process important lines from menu.lst boot entry.
1079                  */
1080                 if (lp->flags == BAM_TITLE) {
1081                         title = strdupa(lp->arg);
1082                 } else if (lp->cmd != NULL) {
1083                         if (strcmp(lp->cmd, "pool_label") == 0) {
 754                         if (strcmp(lp->cmd, "findroot") == 0) {
1084                                 findroot = strdupa(lp->arg);
1085                         } else if (strcmp(lp->cmd, "data_set") == 0) {
 756                         } else if (strcmp(lp->cmd, "bootfs") == 0) {
1086                                 bootfs = strdupa(lp->arg);
1087                         } else if (strcmp(lp->cmd,
1088                             menu_cmds[MODULE_DOLLAR_CMD]) == 0) {
1089                                 if (strstr(lp->arg, "boot_archive") == NULL) {
1090                                         module = strdupa(lp->arg);
1091                                         cvt_hyper_module(module, &kern_path);
1092                                 } else {
1093                                         barchive_path = strdupa(lp->arg);
1094                                 }
1095                         } else if ((strcmp(lp->cmd,
1096                             menu_cmds[KERNEL_DOLLAR_CMD]) == 0) &&
1097                             (cvt_hyper_kernel(lp->arg) < 0)) {
1098                                 ret = BAM_NOCHANGE;
1099                                 goto abort;
1100                         }
1101                 }

1103                 if (lp == ent->end)
1104                         break;
```



```
1105         }

1107         /*
1108          * If findroot, module or kern_path are NULL, the boot entry is
1109          * malformed.
1110          */
1111         if (findroot == NULL) {
1112                 bam_error(FINDROOT_NOT_FOUND, curdef);
1113                 goto abort;
1114         }

1116         if (module == NULL) {
1117                 bam_error(MODULE_NOT_PARSEABLE, curdef);
1118                 goto abort;
1119         }

1121         if (kern_path == NULL) {
1122                 bam_error(KERNEL_NOT_FOUND, curdef);
1123                 goto abort;
1124         }

1126         /*
1127          * Assemble new kernel and module arguments from parsed values.
1128          *
1129          * First, change the kernel directory from the hypervisor version to
1130          * that needed for a metal kernel.
1131          */
1132         newstr = modify_path(kern_path, HYPER_KERNEL_DIR, METAL_KERNEL_DIR);
1133         free(kern_path);
1134         kern_path = newstr;

1136         /* allocate initial space for the kernel path */
1137         len = strlen(kern_path) + 1;
1138         zfslen = (zfs_boot ? (WHITESPC(1) + strlen(ZFS_BOOT)) : 0);

1140         if ((kernel = malloc(len + zfslen)) == NULL) {
1141                 free(kern_path);
1142                 bam_error(NO_MEM, len + zfslen);
1143                 bam_exit(1);
1144         }

1146         (void) snprintf(kernel, len, "%s", kern_path);
1147         free(kern_path);

1149         if (zfs_boot) {
1150                 char *zfsstr = alloca(zfslen + 1);

1152                 (void) snprintf(zfsstr, zfslen + 1, " %s", ZFS_BOOT);
1153                 (void) strcat(kernel, zfsstr);
1154                 emit_bflag = 0;
1155         }

1157         /*
1158          * Process the bootenv.rc file to look for boot options that would be
1159          * the same as what the hypervisor had manually set, as we need not set
1160          * those explicitly.
1161          *
1162          * If there's no bootenv.rc, it's not an issue.
1163          */
1164         parse_bootenvrc(osroot);

1166         /*
1167          * Don't emit a console setting if it's the same as what would be
1168          * set by bootenv.rc.
1169          */
1170         if ((console_dev != NULL) && (bootenv_rc_console == NULL ||
```

```
1171             (strcmp(console_dev, bootenv_rc_console) != 0))) {
1172                 if (emit_bflag) {
1173                         newstr = append_str(kernel, BFLAG, " ");
1174                         free(kernel);
1175                         kernel = append_str(newstr, "console=", " ");
1176                         free(newstr);
1177                         newstr = append_str(kernel, console_dev, "");
1178                         free(kernel);
1179                         kernel = newstr;
1180                         emit_bflag = 0;
1181                 } else {
1182                         newstr = append_str(kernel, "console=", ",");
1183                         free(kernel);
1184                         kernel = append_str(newstr, console_dev, "");
1185                         free(newstr);
1186                 }
1187         }

1189         /*
1190          * We have to do some strange processing here because the hypervisor's
1191          * serial ports default to "9600,8,n,1,-" if "comX=auto" is specified,
1192          * or to "auto" if nothing is specified.
1193          *
1194          * This could result in a serial mode setting string being added when
1195          * it would otherwise not be needed, but it's better to play it safe.
1196          */
1197         if (emit_bflag) {
1198                 newstr = append_str(kernel, BFLAG, " ");
1199                 free(kernel);
1200                 kernel = newstr;
1201                 delim = " ";
1202                 emit_bflag = 0;
1203         }

1205         if ((serial_config[0] != NULL) && (bootenv_rc_serial[0] == NULL ||
1206             (strcmp(serial_config[0], bootenv_rc_serial[0]) != 0))) {
1207                 newstr = append_str(kernel, "ttya-mode='", delim);
1208                 free(kernel);

1210                 /*
1211                  * Pass the serial configuration as the delimiter to
1212                  * append_str() as it will be inserted between the current
1213                  * string and the string we're appending, in this case the
1214                  * closing single quote.
1215                  */
1216                 kernel = append_str(newstr, "'", serial_config[0]);
1217                 free(newstr);
1218                 delim = ",";
1219         }

1221         if ((serial_config[1] != NULL) && (bootenv_rc_serial[1] == NULL ||
1222             (strcmp(serial_config[1], bootenv_rc_serial[1]) != 0))) {
1223                 newstr = append_str(kernel, "ttyb-mode='", delim);
1224                 free(kernel);

1226                 /*
1227                  * Pass the serial configuration as the delimiter to
1228                  * append_str() as it will be inserted between the current
1229                  * string and the string we're appending, in this case the
1230                  * closing single quote.
1231                  */
1232                 kernel = append_str(newstr, "'", serial_config[1]);
1233                 free(newstr);
1234                 delim = ",";
1235         }
```



```
1237         /* shut off warning messages from the entry line parser */
1238         if (ent->flags & BAM_ENTRY_BOOTADM)
1239                 ent->flags &= ~BAM_ENTRY_BOOTADM;

1241         BAM_DPRINTF((D_CVT_CMD_KERN_DOLLAR, fcn, kernel));
1242         BAM_DPRINTF((D_CVT_CMD_MOD_DOLLAR, fcn, module));

1244         if ((newdef = add_boot_entry(mp, title, findroot, kernel, NULL,
1245             barchive_path, bootfs)) == BAM_ERROR) {
1246                 free(kernel);
1247                 return (newdef);
1248         }

1250         /*
1251          * Now try to delete the current default entry from the menu and add
1252          * the new hypervisor entry with the parameters we've setup.
1253          */
1254         if (delete_boot_entry(mp, curdef, DBE_QUIET) == BAM_SUCCESS)
1255                 newdef--;
1256         else
1257                 bam_print(NEW_BOOT_ENTRY, title);

1259         free(kernel);

1261         /*
1262          * If we successfully created the new entry, set the default boot
1263          * entry to that entry and let the caller know the new menu should
1264          * be written out.
1265          */
1266         return (set_global(mp, menu_cmds[DEFAULT_CMD], newdef));

1268 abort:
1269         if (ret != BAM_NOCHANGE)
1270                 bam_error(METAL_ABORT, osroot);

1272         return (ret);
1273 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   17967 Fri Aug 31 05:08:49 2012
new/grub/Makefile.util.def
grub patch
**********************************************************
______unchanged_portion_omitted_

 306 program = {
 307   name = grub-solarislst2cfg;
 308   installdir = sbin;
 309   mansection = 8;
 310   common = util/grub-solarislst2cfg.c;
 311   common = util/ieee1275/ofpath.c;
 312   common = grub-core/kern/emu/argp_common.c;

 314   ldadd = libgrubmods.a;
 315   ldadd = libgrubgcry.a;
 316   ldadd = libgrubkern.a;
 317   ldadd = grub-core/gnulib/libgnu.a;
 318   ldadd = '$(LIBINTL) $(LIBDEVMAPPER) $(LIBUTIL) $(LIBZFS) $(LIBNVPAIR) $(LIBGEO
 319 };

 321 program = {
 322 #endif /* ! codereview */
 323   name = grub-bios-setup;
 324   installdir = sbin;
 325   mansection = 8;
 326   common = util/grub-setup.c;
 327   common = util/lvm.c;
 328   common = grub-core/kern/emu/argp_common.c;
 329   common = grub-core/lib/reed_solomon.c;

 331   ldadd = libgrubmods.a;
 332   ldadd = libgrubkern.a;
 333   ldadd = libgrubgcry.a;
 334   ldadd = grub-core/gnulib/libgnu.a;
 335   ldadd = '$(LIBINTL) $(LIBDEVMAPPER) $(LIBUTIL) $(LIBZFS) $(LIBNVPAIR) $(LIBGEO
 336   cppflags = '-DGRUB_SETUP_BIOS=1';
 337 };

 339 program = {
 340   name = grub-sparc64-setup;
 341   installdir = sbin;
 342   mansection = 8;
 343   common = util/grub-setup.c;
 344   common = util/lvm.c;
 345   common = grub-core/kern/emu/argp_common.c;
 346   common = grub-core/lib/reed_solomon.c;
 347   common = util/ieee1275/ofpath.c;

 349   ldadd = libgrubmods.a;
 350   ldadd = libgrubkern.a;
 351   ldadd = libgrubgcry.a;
 352   ldadd = grub-core/gnulib/libgnu.a;
 353   ldadd = '$(LIBINTL) $(LIBDEVMAPPER) $(LIBUTIL) $(LIBZFS) $(LIBNVPAIR) $(LIBGEO
 354   cppflags = '-DGRUB_SETUP_SPARC64=1';
 355 };

 357 program = {
 358   name = grub-ofpathname;
 359   installdir = sbin;
 360   mansection = 8;
 361   common = util/ieee1275/grub-ofpathname.c;
 362   common = util/ieee1275/ofpath.c;

 364   ldadd = libgrubmods.a;
```



```
 365   ldadd = libgrubgcry.a;
 366   ldadd = libgrubkern.a;
 367   ldadd = grub-core/gnulib/libgnu.a;
 368   ldadd = '$(LIBINTL) $(LIBDEVMAPPER) $(LIBUTIL) $(LIBGEOM)';
 369 };

 371 program = {
 372   name = grub-mklayout;
 373   mansection = 1;

 375   common = util/grub-mklayout.c;
 376   common = grub-core/kern/emu/argp_common.c;

 378   ldadd = libgrubmods.a;
 379   ldadd = libgrubgcry.a;
 380   ldadd = libgrubkern.a;
 381   ldadd = grub-core/gnulib/libgnu.a;
 382   ldadd = '$(LIBINTL) $(LIBDEVMAPPER) $(LIBZFS) $(LIBNVPAIR) $(LIBGEOM)';
 383 };

 385 data = {
 386   common = util/grub.d/README;
 387   installdir = grubconf;
 388 };

 390 script = {
 391   name = '00_header';
 392   common = util/grub.d/00_header.in;
 393   installdir = grubconf;
 394 };

 396 script = {
 397   name = '10_windows';
 398   common = util/grub.d/10_windows.in;
 399   installdir = grubconf;
 400   condition = COND_HOST_WINDOWS;
 401 };

 403 script = {
 404   name = '10_hurd';
 405   common = util/grub.d/10_hurd.in;
 406   installdir = grubconf;
 407   condition = COND_HOST_HURD;
 408 };

 410 script = {
 411   name = '10_kfreebsd';
 412   common = util/grub.d/10_kfreebsd.in;
 413   installdir = grubconf;
 414   condition = COND_HOST_KFREEBSD;
 415 };

 417 script = {
 418   name = '10_illumos';
 419   common = util/grub.d/10_illumos.in;
 420   installdir = grubconf;
 421   condition = COND_HOST_ILLUMOS;
 422 };

 424 script = {
 425   name = '10_netbsd';
 426   common = util/grub.d/10_netbsd.in;
 427   installdir = grubconf;
 428   condition = COND_HOST_NETBSD;
 429 };
```

```
script = {
  name = '10_linux';
  common = util/grub.d/10_linux.in;
  installdir = grubconf;
  condition = COND_HOST_LINUX;
};

script = {
  name = '10_xnu';
  common = util/grub.d/10_xnu.in;
  installdir = grubconf;
  condition = COND_HOST_XNU;
};

script = {
  name = '20_linux_xen';
  common = util/grub.d/20_linux_xen.in;
  installdir = grubconf;
  condition = COND_HOST_LINUX;
};

script = {
  name = '30_os-prober';
  common = util/grub.d/30_os-prober.in;
  installdir = grubconf;
};

script = {
  name = '40_custom';
  common = util/grub.d/40_custom.in;
  installdir = grubconf;
};

script = {
  name = '41_custom';
  common = util/grub.d/41_custom.in;
  installdir = grubconf;
};

script = {
  mansection = 1;
  name = grub-mkrescue;
  x86 = util/grub-mkrescue.in;
  mips_qemu_mips = util/grub-mkrescue.in;
  mips_loongson = util/grub-mkrescue.in;
  ia64_efi = util/grub-mkrescue.in;
  powerpc_ieee1275 = util/powerpc/ieee1275/grub-mkrescue.in;
  enable = i386_pc;
  enable = i386_efi;
  enable = x86_64_efi;
  enable = i386_qemu;
  enable = i386_multiboot;
  enable = i386_coreboot;
  enable = mips_qemu_mips;
  enable = mips_loongson;
  enable = ia64_efi;
  enable = powerpc_ieee1275;
};

script = {
  mansection = 1;
  name = grub-mkstandalone;
  common = util/grub-mkstandalone.in;
};

script = {
```



```
  mansection = 8;
  installdir = sbin;
  name = grub-install;

  common = util/grub-install.in;
  enable = noemu;
};

script = {
  mansection = 8;
  installdir = sbin;
  name = grub-mknetdir;

  common = util/grub-mknetdir.in;
};

script = {
  name = grub-mkconfig;
  common = util/grub-mkconfig.in;
  mansection = 8;
  installdir = sbin;
};

script = {
  name = grub-set-default;
  common = util/grub-set-default.in;
  mansection = 8;
  installdir = sbin;
};

script = {
  name = grub-reboot;
  common = util/grub-reboot.in;
  mansection = 8;
  installdir = sbin;
};

script = {
  name = grub-mkconfig_lib;
  common = util/grub-mkconfig_lib.in;
  installdir = noinst;
};

script = {
  name = grub-kbdcomp;
  common = util/grub-kbdcomp.in;
  mansection = 1;
};

script = {
  name = grub-shell;
  common = tests/util/grub-shell.in;
  installdir = noinst;
};

script = {
  name = grub-shell-tester;
  common = tests/util/grub-shell-tester.in;
  installdir = noinst;
};

script = {
  testcase;
  name = example_scripted_test;
  common = tests/example_scripted_test.in;
};
```

```
564 script = {
565   testcase;
566   name = example_grub_script_test;
567   common = tests/example_grub_script_test.in;
568 };

570 script = {
571   testcase;
572   name = grub_script_echo1;
573   common = tests/grub_script_echo1.in;
574 };

576 script = {
577   testcase;
578   name = grub_script_leading_whitespace;
579   common = tests/grub_script_leading_whitespace.in;
580 };

582 script = {
583   testcase;
584   name = grub_script_echo_keywords;
585   common = tests/grub_script_echo_keywords.in;
586 };

588 script = {
589   testcase;
590   name = grub_script_vars1;
591   common = tests/grub_script_vars1.in;
592 };

594 script = {
595   testcase;
596   name = grub_script_for1;
597   common = tests/grub_script_for1.in;
598 };

600 script = {
601   testcase;
602   name = grub_script_while1;
603   common = tests/grub_script_while1.in;
604 };

606 script = {
607   testcase;
608   name = grub_script_if;
609   common = tests/grub_script_if.in;
610 };

612 script = {
613   testcase;
614   name = grub_script_blanklines;
615   common = tests/grub_script_blanklines.in;
616 };

618 script = {
619   testcase;
620   name = grub_script_final_semicolon;
621   common = tests/grub_script_final_semicolon.in;
622 };

624 script = {
625   testcase;
626   name = grub_script_dollar;
627   common = tests/grub_script_dollar.in;
628 };
```



```
630 script = {
631   testcase;
632   name = grub_script_comments;
633   common = tests/grub_script_comments.in;
634 };

636 script = {
637   testcase;
638   name = grub_script_functions;
639   common = tests/grub_script_functions.in;
640 };

642 script = {
643   testcase;
644   name = grub_script_break;
645   common = tests/grub_script_break.in;
646 };

648 script = {
649   testcase;
650   name = grub_script_continue;
651   common = tests/grub_script_continue.in;
652 };

654 script = {
655   testcase;
656   name = grub_script_shift;
657   common = tests/grub_script_shift.in;
658 };

660 script = {
661   testcase;
662   name = grub_script_blockarg;
663   common = tests/grub_script_blockarg.in;
664 };

666 script = {
667   testcase;
668   name = grub_script_setparams;
669   common = tests/grub_script_setparams.in;
670 };

672 script = {
673   testcase;
674   name = grub_script_return;
675   common = tests/grub_script_return.in;
676 };

678 script = {
679   testcase;
680   name = grub_cmd_regexp;
681   common = tests/grub_cmd_regexp.in;
682 };

684 script = {
685   testcase;
686   name = grub_script_expansion;
687   common = tests/grub_script_expansion.in;
688 };

690 script = {
691   testcase;
692   name = grub_script_not;
693   common = tests/grub_script_not.in;
694 };
```

```
696 script = {
697   testcase;
698   name = partmap_test;
699   common = tests/partmap_test.in;
700 };

702 script = {
703   testcase;
704   name = grub_cmd_echo;
705   common = tests/grub_cmd_echo.in;
706 };

708 script = {
709   testcase;
710   name = grub_script_gettext;
711   common = tests/grub_script_gettext.in;
712 };

714 script = {
715   testcase;
716   name = grub_script_strcmp;
717   common = tests/grub_script_strcmp.in;
718 };

720 program = {
721   testcase;
722   name = example_unit_test;
723   common = tests/example_unit_test.c;
724   common = tests/lib/unit_test.c;
725   common = grub-core/kern/list.c;
726   common = grub-core/kern/misc.c;
727   common = grub-core/tests/lib/test.c;
728   ldadd = libgrubmods.a;
729   ldadd = libgrubgcry.a;
730   ldadd = libgrubkern.a;
731   ldadd = grub-core/gnulib/libgnu.a;
732   ldadd = '$(LIBDEVMAPPER) $(LIBZFS) $(LIBNVPAIR) $(LIBGEOM)';
733 };

735 program = {
736   testcase;
737   name = printf_test;
738   common = tests/printf_unit_test.c;
739   common = tests/lib/unit_test.c;
740   common = grub-core/kern/list.c;
741   common = grub-core/kern/misc.c;
742   common = grub-core/tests/lib/test.c;
743   ldadd = libgrubmods.a;
744   ldadd = libgrubgcry.a;
745   ldadd = libgrubkern.a;
746   ldadd = grub-core/gnulib/libgnu.a;
747   ldadd = '$(LIBDEVMAPPER) $(LIBZFS) $(LIBNVPAIR) $(LIBGEOM)';
748 };

750 program = {
751   testcase;
752   name = cmp_test;
753   common = tests/cmp_unit_test.c;
754   common = tests/lib/unit_test.c;
755   common = grub-core/kern/list.c;
756   common = grub-core/kern/misc.c;
757   common = grub-core/tests/lib/test.c;
758   ldadd = libgrubmods.a;
759   ldadd = libgrubgcry.a;
760   ldadd = libgrubkern.a;
```



```
761   ldadd = grub-core/gnulib/libgnu.a;
762   ldadd = '$(LIBDEVMAPPER) $(LIBZFS) $(LIBNVPAIR) $(LIBGEOM)';
763 };

765 program = {
766   name = grub-menulst2cfg;
767   mansection = 1;
768   common = util/grub-menulst2cfg.c;
769   common = grub-core/lib/legacy_parse.c;
770   common = grub-core/lib/i386/pc/vesa_modes_table.c;

772   ldadd = libgrubmods.a;
773   ldadd = libgrubgcry.a;
774   ldadd = libgrubkern.a;
775   ldadd = grub-core/gnulib/libgnu.a;
776   ldadd = '$(LIBINTL) $(LIBDEVMAPPER) $(LIBZFS) $(LIBNVPAIR) $(LIBGEOM)';
777 };
```

```
**********************************************************
   34261 Fri Aug 31 05:08:49 2012
new/grub/grub-core/Makefile.core.def
grub patch
**********************************************************
______unchanged_portion_omitted_

1486 module = {
1487   name = illumos_entries;
1488   common = commands/illumos_entries.c;
1489 };

1491 module = {
1492   name = solarislegacy;
1493   common = commands/solarislegacy.c;
1494 };

1496 module = {
1497 #endif /* ! codereview */
1498   name = part_acorn;
1499   common = partmap/acorn.c;
1500 };

1502 module = {
1503   name = part_amiga;
1504   common = partmap/amiga.c;
1505 };

1507 module = {
1508   name = part_apple;
1509   common = partmap/apple.c;
1510 };

1512 module = {
1513   name = part_gpt;
1514   common = partmap/gpt.c;
1515 };

1517 module = {
1518   name = part_msdos;
1519   common = partmap/msdos.c;
1520 };

1522 module = {
1523   name = part_sun;
1524   common = partmap/sun.c;
1525 };

1527 module = {
1528   name = part_plan;
1529   common = partmap/plan.c;
1530 };

1532 module = {
1533   name = part_dvh;
1534   common = partmap/dvh.c;
1535 };

1537 module = {
1538   name = part_bsd;
1539   common = partmap/bsdlabel.c;
1540 };

1542 module = {
1543   name = part_sunpc;
1544   common = partmap/sunpc.c;
```



```
1545 };

1547 module = {
1548   name = msdospart;
1549   common = parttool/msdospart.c;
1550 };

1552 module = {
1553   name = at_keyboard;
1554   common = term/at_keyboard.c;
1555   enable = x86;
1556 };

1558 module = {
1559   name = gfxterm;
1560   common = term/gfxterm.c;
1561   enable = videomodules;
1562 };

1564 module = {
1565   name = serial;
1566   common = term/serial.c;
1567   x86 = term/ns8250.c;
1568   ieee1275 = term/ieee1275/serial.c;
1569   efi = term/efi/serial.c;

1571   enable = terminfomodule;
1572   enable = ieee1275;
1573 };

1575 module = {
1576   name = sendkey;
1577   i386_pc = commands/i386/pc/sendkey.c;
1578   enable = i386_pc;
1579 };

1581 module = {
1582   name = terminfo;
1583   common = term/terminfo.c;
1584   common = term/tparm.c;
1585   enable = terminfomodule;
1586 };

1588 module = {
1589   name = usb_keyboard;
1590   common = term/usb_keyboard.c;
1591   enable = usb;
1592 };

1594 module = {
1595   name = vga;
1596   common = video/i386/pc/vga.c;
1597   enable = i386_pc;
1598   enable = i386_coreboot;
1599   enable = i386_multiboot;
1600 };

1602 module = {
1603   name = vga_text;
1604   common = term/i386/pc/vga_text.c;
1605   common = term/i386/vga_common.c;
1606   enable = i386_pc;
1607 };

1609 module = {
1610   name = video_cirrus;
```

```
1611   x86 = video/cirrus.c;
1612   enable = x86;
1613 };

1615 module = {
1616   name = video_bochs;
1617   x86 = video/bochs.c;
1618   enable = x86;
1619 };

1621 module = {
1622   name = functional_test;
1623   common = tests/lib/functional_test.c;
1624   common = tests/lib/test.c;
1625 };

1627 module = {
1628   name = exfctest;
1629   common = tests/example_functional_test.c;
1630 };

1632 module = {
1633   name = bitmap;
1634   common = video/bitmap.c;
1635   enable = videomodules;
1636 };

1638 module = {
1639   name = bitmap_scale;
1640   common = video/bitmap_scale.c;
1641   enable = videomodules;
1642 };

1644 module = {
1645   name = efi_gop;
1646   efi = video/efi_gop.c;
1647   enable = efi;
1648 };

1650 module = {
1651   name = efi_uga;
1652   efi = video/efi_uga.c;
1653   enable = i386_efi;
1654   enable = x86_64_efi;
1655 };

1657 module = {
1658   name = jpeg;
1659   common = video/readers/jpeg.c;
1660 };

1662 module = {
1663   name = png;
1664   common = video/readers/png.c;
1665 };

1667 module = {
1668   name = tga;
1669   common = video/readers/tga.c;
1670 };

1672 module = {
1673   name = vbe;
1674   common = video/i386/pc/vbe.c;
1675   enable = i386_pc;
1676   enable = i386_coreboot;
```



```
1677   enable = i386_multiboot;
1678 };

1680 module = {
1681   name = video_fb;
1682   common = video/fb/video_fb.c;
1683   common = video/fb/fbblit.c;
1684   common = video/fb/fbfill.c;
1685   common = video/fb/fbutil.c;
1686   enable = videomodules;
1687 };

1689 module = {
1690   name = video;
1691   common = video/video.c;
1692   common = video/colors.c;
1693   enable = videomodules;
1694 };

1696 module = {
1697   name = ieee1275_fb;
1698   ieee1275 = video/ieee1275.c;
1699   enable = powerpc_ieee1275;
1700 };

1702 module = {
1703   name = sdl;
1704   emu = video/emu/sdl.c;
1705   enable = emu;
1706   condition = COND_GRUB_EMU_SDL;
1707 };

1709 module = {
1710   name = datehook;
1711   common = hook/datehook.c;
1712 };

1714 module = {
1715   name = net;
1716   common = net/net.c;
1717   common = net/dns.c;
1718   common = net/bootp.c;
1719   common = net/ip.c;
1720   common = net/udp.c;
1721   common = net/tcp.c;
1722   common = net/icmp.c;
1723   common = net/icmp6.c;
1724   common = net/ethernet.c;
1725   common = net/arp.c;
1726   common = net/netbuff.c;
1727 };

1729 module = {
1730   name = tftp;
1731   common = net/tftp.c;
1732 };

1734 module = {
1735   name = http;
1736   common = net/http.c;
1737 };

1739 module = {
1740   name = ofnet;
1741   common = net/drivers/ieee1275/ofnet.c;
1742   enable = ieee1275;
```

```
1743 };

1745 module = {
1746   name = efinet;
1747   common = net/drivers/efi/efinet.c;
1748   enable = efi;
1749 };

1751 module = {
1752   name = emunet;
1753   emu = net/drivers/emu/emunet.c;
1754   enable = emu;
1755 };

1757 module = {
1758   name = legacycfg;
1759   common = commands/legacycfg.c;
1760   common = lib/legacy_parse.c;
1761   emu = lib/i386/pc/vesa_modes_table.c;
1762   enable = i386_pc;
1763   enable = emu;
1764 };

1766 module = {
1767   name = test_blockarg;
1768   common = tests/test_blockarg.c;
1769 };

1771 module = {
1772   name = xzio;
1773   common = io/xzio.c;
1774   common = lib/xzembed/xz_dec_bcj.c;
1775   common = lib/xzembed/xz_dec_lzma2.c;
1776   common = lib/xzembed/xz_dec_stream.c;
1777   cppflags = '-I$(srcdir)/lib/posix_wrap -I$(srcdir)/lib/xzembed';
1778   cflags='-Wno-unreachable-code';
1779 };

1781 module = {
1782   name = lzopio;
1783   common = io/lzopio.c;
1784   common = lib/minilzo/minilzo.c;
1785   cflags = '$(CFLAGS_POSIX) -Wno-undef -Wno-redundant-decls -Wno-error';
1786   cppflags = '-I$(srcdir)/lib/posix_wrap -I$(srcdir)/lib/minilzo -DMINILZO_HAVE_
1787 };

1789 module = {
1790   name = testload;
1791   common = commands/testload.c;
1792 };

1794 module = {
1795   name = backtrace;
1796   x86 = lib/i386/backtrace.c;
1797   common = lib/backtrace.c;
1798   enable = x86;
1799 };

1801 module = {
1802   name = lsapm;
1803   common = commands/i386/pc/lsapm.c;
1804   enable = i386_pc;
1805 };

1807 module = {
1808   name = keylayouts;
```



```
1809   common = commands/keylayouts.c;
1810   enable = videomodules;
1811 };

1813 module = {
1814   name = priority_queue;
1815   common = lib/priority_queue.c;
1816 };

1818 module = {
1819   name = time;
1820   common = commands/time.c;
1821 };

1823 module = {
1824   name = cacheinfo;
1825   common = commands/cacheinfo.c;
1826   condition = COND_ENABLE_CACHE_STATS;
1827 };

1829 module = {
1830   name = adler32;
1831   common = lib/adler32.c;
1832 };

1834 module = {
1835   name = crc64;
1836   common = lib/crc64.c;
1837 };

1839 module = {
1840   name = all_video;
1841   common = lib/fake_module.c;
1842 };

1844 module = {
1845   name = gdb;
1846   common = gdb/cstub.c;
1847   common = gdb/gdb.c;
1848   i386 = gdb/i386/idt.c;
1849   i386 = gdb/i386/machdep.S;
1850   i386 = gdb/i386/signal.c;
1851   enable = i386;
1852 };
```

```
**********************************************************
    4544 Fri Aug 31 05:08:50 2012
new/grub/grub-core/commands/illumos_entries.c
grub patch
**********************************************************
```

```
/*
 *  GRUB  --  GRand Unified Bootloader
 *  Copyright (C) 2012 Daniil Lunev
 *
 *  GRUB is free software: you can redistribute it and/or modify
 *  it under the terms of the GNU General Public License as published by
 *  the Free Software Foundation, either version 3 of the License, or
 *  (at your option) any later version.
 *
 *  GRUB is distributed in the hope that it will be useful,
 *  but WITHOUT ANY WARRANTY; without even the implied warranty of
 *  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 *  GNU General Public License for more details.
 *
 *  You should have received a copy of the GNU General Public License
 *  along with GRUB.  If not, see <http://www.gnu.org/licenses/>.
 */

#include <grub/types.h>
#include <grub/file.h>
#include <grub/disk.h>
#include <grub/misc.h>
#include <grub/err.h>
#include <grub/dl.h>
#include <grub/extcmd.h>
#include <grub/i18n.h>
#include <grub/normal.h>

#include "menu_managing.c"

GRUB_MOD_LICENSE ("GPLv3+");

static const struct grub_arg_option options[] = {
  {0, 0, 0, 0, 0, 0}
};

static const char * globals[] = {
  "default_entry",
  "timeout",
  "serial",
  "terminal",
  NULL
};

static grub_err_t
get_value(char * buf, char ** value)
{
  *value = grub_strchr(buf, '=');
  if (! *value)
    return grub_error (GRUB_ERR_INVALID_COMMAND, N_("illumos syntax error"));
  **value = '\0';
  ++(*value);
  return 0;
}

static int
check_param(char * param)
{
  int i = 0;

  for (; params_list[i]; ++i)
    if (!grub_strcmp(param, params_list[i]))
      return i;

  return -1;
}

static int
check_global(char * param)
{
  int i = 0;

  for (; globals[i]; ++i)
    if (!grub_strcmp(param, globals[i]))
      return i;

  return -1;
}

static grub_err_t
parse_config(grub_file_t file, entries * entry_list)
{
  char * param;
  char * value;
  entries * current_entry = NULL;
  int param_id = 0;
  int global_id = 0;
  grub_err_t err;

  for(;;) {
    param = grub_file_getline(file);
    if (! param)
      return grub_errno;

    if ((*param == '#') || (*param == ' ') ||
      (*param == '\t') || (*param == '\n') ||
      (*param == 0)) {
      grub_free(param);
      continue;
    }
    global_id = -1;

    err = get_value(param, &value);
    if (err)
      return err;

    param_id = check_param(param);
    if (param_id < 0) {
      global_id = check_global(param);
      if (global_id < 0) {
        grub_free(param);
        return grub_error (GRUB_ERR_INVALID_COMMAND, N_("illumos syntax error"));
      }
    }
    if (err)
      return err;
    if (!value[0])
      return grub_error (GRUB_ERR_INVALID_COMMAND, N_("illumos syntax error"));
    if (global_id < 0) {
      if (param_id == 0) {
        current_entry = new_entry(value, entry_list);
        if (! current_entry) {
          grub_free(param);
          return grub_error (GRUB_ERR_OUT_OF_MEMORY, N_("memory can not be alloc
        }
      } else {
        grub_strcpy(current_entry->entry_info[param_id], value);
```

```
128       }
129     } else {
130       char line[512];
131       switch (global_id) {
132       case 0:
133         grub_env_set("default", value);
134         break;
135       case 1:
136         grub_env_set("timeout", value);
137         break;
138       case 2:
139         grub_strcpy(line, "serial ");
140         grub_strcat(line, value);
141         grub_normal_parse_line(line, NULL);
142         break;
143       case 3:
144         grub_strcpy(line, "terminal_input ");
145         grub_strcat(line, value);
146         grub_strcat(line, "; terminal_output ");
147         grub_strcat(line, value);
148         grub_normal_parse_line(line, NULL);
149         break;
150       default:
151         break;
152       }
153     }
154     grub_free(param);
155   }
156   return 0;
157 }

159 static grub_err_t
160 grub_cmd_illumos_entries (grub_extcmd_context_t ctxt __attribute__ ((unused)), i
161 {
162   grub_file_t file;
163   entries * menu_entries = NULL;
164
165   if (argc != 1)
166     return grub_error (GRUB_ERR_BAD_ARGUMENT, N_("filename expected"));
167
168   file = grub_file_open(args[0]);
169
170   if (!file)
171     return grub_errno;
172
173   init_entries(&menu_entries);
174
175   if (parse_config(file, menu_entries) == 0) {
176     add_entries(menu_entries);
177   }
178   clear_entries(menu_entries);
179
180   grub_refresh();
181   grub_file_close(file);

183   return 0;
184 }

186 static grub_extcmd_t illumos_entries;

188 GRUB_MOD_INIT(illumos_entries)
189 {
190   illumos_entries = grub_register_extcmd ("illumos_entries", grub_cmd_illumos_en
191             N_("FILE"), N_("Define an illumos menu entries."), options);
192 }
```



```
194 GRUB_MOD_FINI(illumos_entries)
195 {
196   grub_unregister_extcmd (illumos_entries);
197 }
198 #endif /* ! codereview */
```

```
**********************************************************
    5949 Fri Aug 31 05:08:50 2012
new/grub/grub-core/commands/menu_managing.c
grub patch
**********************************************************
```

```
/*
 *  GRUB  --  GRand Unified Bootloader
 *  Copyright (C) 2012 Daniil Lunev
 *
 *  GRUB is free software: you can redistribute it and/or modify
 *  it under the terms of the GNU General Public License as published by
 *  the Free Software Foundation, either version 3 of the License, or
 *  (at your option) any later version.
 *
 *  GRUB is distributed in the hope that it will be useful,
 *  but WITHOUT ANY WARRANTY; without even the implied warranty of
 *  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 *  GNU General Public License for more details.
 *
 *  You should have received a copy of the GNU General Public License
 *  along with GRUB.  If not, see <http://www.gnu.org/licenses/>.
 */

#define VALUE_SIZE 512

typedef struct entries entries;
struct entries {
  entries *next;
  char ** entry_info;
};

enum param_consts {
  ENTRY_NAME,
  POOL_UUID,
  POOL_LABEL,
  DATA_SET,
  KERNEL_PATH,
  KERNEL_OPTIONS,
  BA_PATH,
  DOLLAR_KERNEL_PATH,
  DOLLAR_BA_PATH,
};

static const char * params_list[] = {
  "entry_name",
  "pool_uuid",
  "pool_label",
  "data_set",
  "kernel_path",
  "kernel_options",
  "module",
  "kernel_path$",
  "module$",
  NULL
};

static grub_err_t
init_entries(entries ** menu_entries)
{
  (*menu_entries) = (entries*) grub_zalloc(sizeof (menu_entries));
  if (! *menu_entries)
    return grub_error (GRUB_ERR_OUT_OF_MEMORY, N_("memory can not be allocated")
  return 0;
}

static entries *
```



```
new_entry(char * name, entries * entry_list)
{
  unsigned int i;

  if (entry_list->next)
    do {
      entry_list = entry_list->next;
      if (!grub_strcmp(name, entry_list->entry_info[0]))
        return entry_list;
    } while (entry_list->next);
  entry_list->next = (entries*) grub_zalloc(sizeof(entries));
  if (! entry_list->next)
    return NULL;
  entry_list = entry_list->next;
  entry_list->entry_info = (char**) grub_zalloc(sizeof(params_list));
  for (i = 0; i < sizeof(params_list) / sizeof(*params_list); i++) {
    entry_list->entry_info[i] = (char *) grub_zalloc(VALUE_SIZE);
    if (! entry_list->entry_info[i])
      return NULL;
  }
  grub_strcpy(entry_list->entry_info[0], name);
  return entry_list;
}

static void
clear_entries(entries * entry_list)
{
  entries * next;
  unsigned int i;

  next = entry_list->next;
  grub_free(entry_list);
  entry_list = next;

  while (entry_list) {
    next = entry_list->next;
    if (entry_list->entry_info) {
      for (i = 0; i < sizeof(params_list) / sizeof(*params_list); i++)
        if (entry_list->entry_info[i])
          grub_free(entry_list->entry_info[i]);
      grub_free(entry_list->entry_info);
    }
    grub_free(entry_list);
    entry_list = next;
  }
}

static grub_err_t
add_entries(entries * menu_entries)
{
  grub_err_t err;
  char cl1[] = "os";
  char cl2[] = "illumos";
  char * class[] = {cl1, cl2, NULL};
  char * argv[] = { NULL, NULL };
  char id[512];
  char * entry_source = NULL;
  char * data_set;
  char entry_template[] =
    "insmod part_sunpc\n"
    "insmod part_msdos\n"
    "insmod zfs\n"
    "insmod gzio\n"
    "if cpuid -l ; then\n"
    "  ISADIR=amd64\n"
    "else\n"
```

```
128     "  ISADIR=\n"
129     "fi\n";

131   menu_entries = menu_entries->next;
132   while (menu_entries) {
133     argv[0] = grub_strdup(menu_entries->entry_info[ENTRY_NAME]);
134     if (!argv[0])
135       return grub_error (GRUB_ERR_OUT_OF_MEMORY, N_("memory can not be allocated
136     entry_source = (char*) grub_zalloc(2048);

138     if (!entry_source)
139       return grub_error (GRUB_ERR_OUT_OF_MEMORY, N_("memory can not be allocated

141     if (menu_entries->entry_info[DATA_SET][0] != 0)
142       data_set = grub_strchr(menu_entries->entry_info[DATA_SET], '/');
143     else
144       data_set = grub_strdup("$ZFS_DATASET");
145     grub_strcpy(entry_source, entry_template);
146     if (menu_entries->entry_info[POOL_UUID][0]) {
147       grub_strcat(entry_source, "search --no-floppy --zfs-mirror --fs-uuid --set
148       grub_strcat(entry_source, menu_entries->entry_info[POOL_UUID]);
149     } else {
150       grub_strcat(entry_source, "search --no-floppy --zfs-mirror --label --set=r
151       grub_strcat(entry_source, menu_entries->entry_info[POOL_LABEL]);
152     }
153     grub_strcat(entry_source, "\n");
154     grub_strcat(entry_source, "zfs-bootfs ($root)");
155     if (menu_entries->entry_info[DATA_SET][0] != 0) {
156       grub_strcat(entry_source, data_set);
157       grub_strcat(entry_source, " ZFS_BOOTFS\n");
158     } else {
159       grub_strcat(entry_source, "default ZFS_BOOTFS ZFS_DATASET\n");
160     }
161     grub_strcat(entry_source, "multiboot ($root)");
162     if (! menu_entries->entry_info[DOLLAR_KERNEL_PATH][0]) {
163       grub_strcat(entry_source, data_set);
164       grub_strcat(entry_source, "/@");
165       grub_strcat(entry_source, menu_entries->entry_info[KERNEL_PATH]);
166       grub_strcat(entry_source, " ");
167       grub_strcat(entry_source, menu_entries->entry_info[KERNEL_PATH]);
168     } else {
169       grub_strcat(entry_source, menu_entries->entry_info[DOLLAR_KERNEL_PATH]);
170     }
171     grub_strcat(entry_source, " ");
172     grub_strcat(entry_source, menu_entries->entry_info[KERNEL_OPTIONS]);
173     grub_strcat(entry_source, "\n");
174     grub_strcat(entry_source, "module ($root)");
175     if (! menu_entries->entry_info[DOLLAR_KERNEL_PATH][0]) {
176       grub_strcat(entry_source, data_set);
177       grub_strcat(entry_source, "/@");
178       grub_strcat(entry_source, menu_entries->entry_info[BA_PATH]);
179       grub_strcat(entry_source, " ");
180       grub_strcat(entry_source, menu_entries->entry_info[BA_PATH]);
181     } else {
182       grub_strcat(entry_source, menu_entries->entry_info[DOLLAR_BA_PATH]);
183     }

185     grub_strcpy(id, argv[0]);
186     grub_strcat(id, "-");
187     grub_strcat(id, menu_entries->entry_info[POOL_UUID]);

189     err = grub_normal_add_menu_entry (2, (const char **) argv, class, id,
190                 NULL,  NULL,  NULL, entry_source, 0);

192     if (err)
193       return err;
```



```
194
195     grub_free(argv[0]);
196     menu_entries = menu_entries->next;
197   }
198   grub_free(entry_source);
199   return 0;
200 }
201 #endif /* ! codereview */
```

```
**********************************************************
    7395 Fri Aug 31 05:08:51 2012
new/grub/grub-core/commands/search.c
fixes + mirror
**********************************************************
```

```
/* search.c - search devices based on a file or a filesystem label */
/*
 *  GRUB  --  GRand Unified Bootloader
 *  Copyright (C) 2005,2007,2008,2009  Free Software Foundation, Inc.
 *
 *  GRUB is free software: you can redistribute it and/or modify
 *  it under the terms of the GNU General Public License as published by
 *  the Free Software Foundation, either version 3 of the License, or
 *  (at your option) any later version.
 *
 *  GRUB is distributed in the hope that it will be useful,
 *  but WITHOUT ANY WARRANTY; without even the implied warranty of
 *  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 *  GNU General Public License for more details.
 *
 *  You should have received a copy of the GNU General Public License
 *  along with GRUB.  If not, see <http://www.gnu.org/licenses/>.
 */

#include <grub/types.h>
#include <grub/misc.h>
#include <grub/mm.h>
#include <grub/err.h>
#include <grub/dl.h>
#include <grub/zfs/zfs.h>
#endif /* ! codereview */
#include <grub/device.h>
#include <grub/file.h>
#include <grub/env.h>
#include <grub/command.h>
#include <grub/search.h>
#include <grub/i18n.h>
#include <grub/disk.h>
#include <grub/partition.h>

GRUB_MOD_LICENSE ("GPLv3+");

struct cache_entry
{
  struct cache_entry *next;
  char *key;
  char *value;
};

static struct cache_entry *cache;

void
FUNC_NAME (const char *key, const char *var, int no_floppy,
           char **hints, unsigned nhints, int mirror)
           char **hints, unsigned nhints)
{
  int count = 0;
  int is_cache = 0;
  grub_uint64_t txg = 0;
#endif /* ! codereview */
  grub_fs_autoload_hook_t saved_autoload;

  auto int iterate_device (const char *name);
  int iterate_device (const char *name)
  {
    int found = 0;
```



```
    /* Skip floppy drives when requested.  */
    if (no_floppy &&
        name[0] == 'f' && name[1] == 'd' && name[2] >= '0' && name[2] <= '9')
      return 0;

#ifdef DO_SEARCH_FS_UUID
#define compare_fn grub_strcasecmp
#else
#define compare_fn grub_strcmp
#endif

#ifdef DO_SEARCH_FILE
      {
        char *buf;
        grub_file_t file;

        buf = grub_xasprintf ("(%s)%s", name, key);
        if (! buf)
          return 1;

        grub_file_filter_disable_compression ();
        file = grub_file_open (buf);
        if (file)
          {
            found = 1;
            grub_file_close (file);
          }
        grub_free (buf);
      }
#else
      {
        /* SEARCH_FS_UUID or SEARCH_LABEL */
        grub_device_t dev;
        grub_fs_t fs;
        char *quid;

        dev = grub_device_open (name);
        if (dev)
          {
            fs = grub_fs_probe (dev);

#ifdef DO_SEARCH_FS_UUID
#define read_fn uuid
#else
#define read_fn label
#endif

            if (fs && fs->read_fn)
              {
                fs->read_fn (dev, &quid);

                if (grub_errno == GRUB_ERR_NONE && quid)
                  {
                    if (compare_fn (quid, key) == 0)
                      found = 1;

                    grub_free (quid);
                  }
              }

            grub_device_close (dev);
          }
      }
#endif
```

```
127     if (!is_cache && found && count == 0)
128       {
129         struct cache_entry *cache_ent;
130         cache_ent = grub_malloc (sizeof (*cache_ent));
131         if (cache_ent)
132           {
133             cache_ent->key = grub_strdup (key);
134             cache_ent->value = grub_strdup (name);
135             if (cache_ent->value && cache_ent->key)
136               {
137                 cache_ent->next = cache;
138                 cache = cache_ent;
139               }
140             else
141               {
142                 grub_free (cache_ent->value);
143                 grub_free (cache_ent->key);
144                 grub_free (cache_ent);
145                 grub_errno = GRUB_ERR_NONE;
146               }
147           }
148         else
149           grub_errno = GRUB_ERR_NONE;
150       }

152     if (found)
153       {
154         count++;
155         if (var) {
156           if (! mirror) {
157             grub_env_set (var, name);
158           } else {
159             grub_uint64_t tmp_txg = 0;
160             char * nvlist = NULL;
161             grub_device_t dev = grub_device_open (name);
162             if (! dev) {
163               grub_errno = GRUB_ERR_BAD_DEVICE;
164               return 0;
165             }
166             grub_errno = grub_zfs_fetch_nvlist (dev, &nvlist);
167             grub_device_close (dev);
168             if (grub_errno)
169               return 0;
170             grub_zfs_nvlist_lookup_uint64 (nvlist, ZPOOL_CONFIG_POOL_TXG, &tmp_t
171             if (tmp_txg > txg) {
 29         if (var)
172               grub_env_set (var, name);
173               txg = tmp_txg;
174             }
175             grub_free (nvlist);
176           }
177         } else {
 31         else
178           grub_printf (" %s", name);
179         }
180       }
181 #endif /* ! codereview */

183     grub_errno = GRUB_ERR_NONE;
184     return (found && var && !mirror);
 34     return (found && var);
185   }

187   auto int part_hook (grub_disk_t disk, const grub_partition_t partition);
188   int part_hook (grub_disk_t disk, const grub_partition_t partition)
189   {
```



```
190     char *partition_name, *devname;
191     int ret;

193     partition_name = grub_partition_get_name (partition);
194     if (! partition_name)
195       return 1;

197     devname = grub_xasprintf ("%s,%s", disk->name, partition_name);
198     grub_free (partition_name);
199     if (!devname)
200       return 1;
201     ret = iterate_device (devname);
202     grub_free (devname);

204     return ret;
205   }

207   auto void try (void);
208   void try (void)
209   {
210     unsigned i;
211     struct cache_entry **prev;
212     struct cache_entry *cache_ent;

214     for (prev = &cache, cache_ent = *prev; cache_ent;
215          prev = &cache_ent->next, cache_ent = *prev)
216       if (compare_fn (cache_ent->key, key) == 0)
217         break;
218     if (cache_ent)
219       {
220         is_cache = 1;
221         if (iterate_device (cache_ent->value))
222           {
223             is_cache = 0;
224             return;
225           }
226         is_cache = 0;
227         /* Cache entry was outdated. Remove it.  */
228         if (!count)
229           {
230             grub_free (cache_ent->key);
231             grub_free (cache_ent->value);
232             grub_free (cache_ent);
233             *prev = cache_ent->next;
234           }
235       }

237     for (i = 0; i < nhints; i++)
238       {
239         char *end;
240         if (!hints[i][0])
241           continue;
242         end = hints[i] + grub_strlen (hints[i]) - 1;
243         if (*end == ',')
244           *end = 0;
245         if (iterate_device (hints[i]))
246           {
247             if (!*end)
248               *end = ',';
249             return;
250           }
251         if (!*end)
252           {
253             grub_device_t dev;
254             int ret;
255             dev = grub_device_open (hints[i]);
```

```
256             if (!dev)
257               {
258                 if (!*end)
259                   *end = ',';
260                 continue;
261               }
262             if (!dev->disk)
263               {
264                 grub_device_close (dev);
265                 if (!*end)
266                   *end = ',';
267                 continue;
268               }
269             ret = grub_partition_iterate (dev->disk, part_hook);
270             if (!*end)
271               *end = ',';
272             grub_device_close (dev);
273             if (ret)
274               return;
275           }
276       }
277     grub_device_iterate (iterate_device);
278   }

280   /* First try without autoloading if we're setting variable. */
281   if (var)
282     {
283       saved_autoload = grub_fs_autoload_hook;
284       grub_fs_autoload_hook = 0;
285       try ();

287       /* Restore autoload hook.  */
288       grub_fs_autoload_hook = saved_autoload;

290       /* Retry with autoload if nothing found.  */
291       if (grub_errno == GRUB_ERR_NONE && count == 0)
292         try ();
293     }
294   else
295     try ();

297   if (grub_errno == GRUB_ERR_NONE && count == 0)
298     grub_error (GRUB_ERR_FILE_NOT_FOUND, "no such device: %s", key);
299 }

301 static grub_err_t
302 grub_cmd_do_search (grub_command_t cmd __attribute__ ((unused)), int argc,
303                     char **args)
304 {
305   if (argc == 0)
306     return grub_error (GRUB_ERR_BAD_ARGUMENT, N_("one argument expected"));

308   FUNC_NAME (args[0], argc == 1 ? 0 : args[1], 0, (args + 2),
309              argc > 2 ? argc - 2 : 0);
159              argc > 2 ? argc - 2 : 0);

311   return grub_errno;
312 }
```

*unchanged_portion_omitted_*

```
**********************************************************
    6857 Fri Aug 31 05:08:51 2012
new/grub/grub-core/commands/search_wrap.c
fixes + mirror
**********************************************************
```

```
/* search.c - search devices based on a file or a filesystem label */
/*
 *  GRUB  --  GRand Unified Bootloader
 *  Copyright (C) 2005,2007,2008,2009  Free Software Foundation, Inc.
 *
 *  GRUB is free software: you can redistribute it and/or modify
 *  it under the terms of the GNU General Public License as published by
 *  the Free Software Foundation, either version 3 of the License, or
 *  (at your option) any later version.
 *
 *  GRUB is distributed in the hope that it will be useful,
 *  but WITHOUT ANY WARRANTY; without even the implied warranty of
 *  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 *  GNU General Public License for more details.
 *
 *  You should have received a copy of the GNU General Public License
 *  along with GRUB.  If not, see <http://www.gnu.org/licenses/>.
 */

#include <grub/types.h>
#include <grub/misc.h>
#include <grub/mm.h>
#include <grub/err.h>
#include <grub/dl.h>
#include <grub/env.h>
#include <grub/extcmd.h>
#include <grub/search.h>
#include <grub/i18n.h>

GRUB_MOD_LICENSE ("GPLv3+");

static const struct grub_arg_option options[] =
  {
    {"file",            'f', 0, N_("Search devices by a file."), 0, 0},
    {"label",           'l', 0, N_("Search devices by a filesystem label."),
     0, 0},
    {"fs-uuid",         'u', 0, N_("Search devices by a filesystem UUID."),
     0, 0},
    {"set",             's', GRUB_ARG_OPTION_OPTIONAL,
     N_("Set a variable to the first device found."), N_("VARNAME"),
     ARG_TYPE_STRING},
    {"no-floppy",       'n', 0, N_("Do not probe any floppy drive."), 0, 0},
    {"hint",            'h', GRUB_ARG_OPTION_REPEATABLE,
     N_("First try the device HINT. If HINT ends in comma, "
        "also try subpartitions"), N_("HINT"), ARG_TYPE_STRING},
    {"hint-ieee1275",   0, GRUB_ARG_OPTION_REPEATABLE,
     N_("First try the device HINT if currently running on IEEE1275. "
        "If HINT ends in comma, also try subpartitions"),
     N_("HINT"), ARG_TYPE_STRING},
    {"hint-bios",   0, GRUB_ARG_OPTION_REPEATABLE,
     N_("First try the device HINT if currently running on BIOS. "
        "If HINT ends in comma, also try subpartitions"),
     N_("HINT"), ARG_TYPE_STRING},
    {"hint-baremetal",   0, GRUB_ARG_OPTION_REPEATABLE,
     N_("First try the device HINT if direct hardware access is supported. "
        "If HINT ends in comma, also try subpartitions"),
     N_("HINT"), ARG_TYPE_STRING},
    {"hint-efi",   0, GRUB_ARG_OPTION_REPEATABLE,
     N_("First try the device HINT if currently running on EFI. "
        "If HINT ends in comma, also try subpartitions"),
     N_("HINT"), ARG_TYPE_STRING},
```



```
    {"hint-arc",   0, GRUB_ARG_OPTION_REPEATABLE,
     N_("First try the device HINT if currently running on ARC."
        " If HINT ends in comma, also try subpartitions"),
     N_("HINT"), ARG_TYPE_STRING},
    {"zfs-mirror", 'z', 0, N_("Handle zfs-mirror disk"), 0, 0},
#endif /* ! codereview */
    {0, 0, 0, 0, 0, 0}
  };

enum options
  {
    SEARCH_FILE,
    SEARCH_LABEL,
    SEARCH_FS_UUID,
    SEARCH_SET,
    SEARCH_NO_FLOPPY,
    SEARCH_HINT,
    SEARCH_HINT_IEEE1275,
    SEARCH_HINT_BIOS,
    SEARCH_HINT_BAREMETAL,
    SEARCH_HINT_EFI,
    SEARCH_HINT_ARC,
    SEARCH_ZFS_MIRROR,
#endif /* ! codereview */
  };

static grub_err_t
grub_cmd_search (grub_extcmd_context_t ctxt, int argc, char **args)
{
  struct grub_arg_list *state = ctxt->state;
  const char *var = 0;
  const char *id = 0;
  int i = 0, j = 0, nhints = 0;
  char **hints = NULL;
  int mirror_mode = 0;
#endif /* ! codereview */

  if (state[SEARCH_HINT].set)
    for (i = 0; state[SEARCH_HINT].args[i]; i++)
      nhints++;

#ifdef GRUB_MACHINE_IEEE1275
  if (state[SEARCH_HINT_IEEE1275].set)
    for (i = 0; state[SEARCH_HINT_IEEE1275].args[i]; i++)
      nhints++;
#endif

#ifdef GRUB_MACHINE_EFI
  if (state[SEARCH_HINT_EFI].set)
    for (i = 0; state[SEARCH_HINT_EFI].args[i]; i++)
      nhints++;
#endif

#ifdef GRUB_MACHINE_PCBIOS
  if (state[SEARCH_HINT_BIOS].set)
    for (i = 0; state[SEARCH_HINT_BIOS].args[i]; i++)
      nhints++;
#endif

#ifdef GRUB_MACHINE_ARC
  if (state[SEARCH_HINT_ARC].set)
    for (i = 0; state[SEARCH_HINT_ARC].args[i]; i++)
      nhints++;
#endif

  if (state[SEARCH_HINT_BAREMETAL].set)
```

```
128     for (i = 0; state[SEARCH_HINT_BAREMETAL].args[i]; i++)
129       nhints++;

131   hints = grub_malloc (sizeof (hints[0]) * nhints);
132   if (!hints)
133     return grub_errno;
134   j = 0;

136   if (state[SEARCH_HINT].set)
137     for (i = 0; state[SEARCH_HINT].args[i]; i++)
138       hints[j++] = state[SEARCH_HINT].args[i];

140 #ifdef GRUB_MACHINE_IEEE1275
141   if (state[SEARCH_HINT_IEEE1275].set)
142     for (i = 0; state[SEARCH_HINT_IEEE1275].args[i]; i++)
143       hints[j++] = state[SEARCH_HINT_IEEE1275].args[i];
144 #endif

146 #ifdef GRUB_MACHINE_EFI
147   if (state[SEARCH_HINT_EFI].set)
148     for (i = 0; state[SEARCH_HINT_EFI].args[i]; i++)
149       hints[j++] = state[SEARCH_HINT_EFI].args[i];
150 #endif

152 #ifdef GRUB_MACHINE_ARC
153   if (state[SEARCH_HINT_ARC].set)
154     for (i = 0; state[SEARCH_HINT_ARC].args[i]; i++)
155       hints[j++] = state[SEARCH_HINT_ARC].args[i];
156 #endif

158 #ifdef GRUB_MACHINE_PCBIOS
159   if (state[SEARCH_HINT_BIOS].set)
160     for (i = 0; state[SEARCH_HINT_BIOS].args[i]; i++)
161       hints[j++] = state[SEARCH_HINT_BIOS].args[i];
162 #endif

164   if (state[SEARCH_HINT_BAREMETAL].set)
165     for (i = 0; state[SEARCH_HINT_BAREMETAL].args[i]; i++)
166       hints[j++] = state[SEARCH_HINT_BAREMETAL].args[i];

168   /* Skip hints for future platforms.  */
169   for (j = 0; j < argc; j++)
170     if (grub_memcmp (args[j], "--hint-", sizeof ("--hint-") - 1) != 0)
171       break;

173   if (state[SEARCH_SET].set)
174     var = state[SEARCH_SET].arg ? state[SEARCH_SET].arg : "root";

176   if (argc != j)
177     id = args[j];
178   else if (state[SEARCH_SET].set && state[SEARCH_SET].arg)
179     {
180       id = state[SEARCH_SET].arg;
181       var = "root";
182     }
183   else
184     return grub_error (GRUB_ERR_BAD_ARGUMENT, N_("one argument expected"));

186   if (state[SEARCH_ZFS_MIRROR].set)
187     mirror_mode = 1;

189 #endif /* ! codereview */
190   if (state[SEARCH_LABEL].set)
191     grub_search_label (id, var, state[SEARCH_NO_FLOPPY].set,
192                        hints, nhints, mirror_mode);
 66                        hints, nhints);
```



```
193   else if (state[SEARCH_FS_UUID].set)
194     grub_search_fs_uuid (id, var, state[SEARCH_NO_FLOPPY].set,
195                          hints, nhints, mirror_mode);
 69                          hints, nhints);
196   else if (state[SEARCH_FILE].set)
197     grub_search_fs_file (id, var, state[SEARCH_NO_FLOPPY].set,
198                          hints, nhints, mirror_mode);
 72                          hints, nhints);
199   else
200     return grub_error (GRUB_ERR_INVALID_COMMAND, "unspecified search type");

202   return grub_errno;
203 }
```

*unchanged_portion_omitted_*

```
**********************************************************
    5686 Fri Aug 31 05:08:52 2012
new/grub/grub-core/commands/solarislegacy.c
grub patch
**********************************************************
```

```
/*
 *  GRUB  --  GRand Unified Bootloader
 *  Copyright (C) 2012 Daniil Lunev
 *
 *  GRUB is free software: you can redistribute it and/or modify
 *  it under the terms of the GNU General Public License as published by
 *  the Free Software Foundation, either version 3 of the License, or
 *  (at your option) any later version.
 *
 *  GRUB is distributed in the hope that it will be useful,
 *  but WITHOUT ANY WARRANTY; without even the implied warranty of
 *  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 *  GNU General Public License for more details.
 *
 *  You should have received a copy of the GNU General Public License
 *  along with GRUB.  If not, see <http://www.gnu.org/licenses/>.
 */

#include <grub/types.h>
#include <grub/misc.h>
#include <grub/command.h>
#include <grub/mm.h>
#include <grub/err.h>
#include <grub/dl.h>
#include <grub/file.h>
#include <grub/normal.h>
#include <grub/script_sh.h>
#include <grub/i18n.h>
#include <grub/term.h>
#include <grub/legacy_parse.h>
#include <grub/crypto.h>
#include <grub/auth.h>
#include <grub/disk.h>
#include <grub/partition.h>

#include "menu_managing.c"

GRUB_MOD_LICENSE ("GPLv3+");

static entries *
parse_entries(grub_file_t file)
{
  entries * list = NULL;
  entries * current = NULL;
  init_entries(&list);
  for(;;) {
    char * buf = grub_file_getline(file);
    char * param = buf;
    char * value;

    if (! buf)
      break;

    if ((buf[0] == '#') || (buf[0] == ' ') ||
      (buf[0] == '\t') || (buf[0] == '\n')) {
      grub_free(buf);
      continue;
    }

    while ((*param == ' ') || (*param == '\t'))
      ++param;
```



```
    value = grub_strchr(param, ' ');
    if (! value)
      value = grub_strchr(param, '\t');

    if (! value) {
      grub_free(buf);
      continue;
    }

    *value++ = 0;
    while ((*value == ' ') || (*value == '\t'))
      ++value;

    if ((*value == '\0') || (*value == '\n')) {
      grub_free(buf);
      continue;
    }

    if (! grub_strcmp(param, "default")) {
    } else if (! grub_strcmp(param, "timeout")) {
    } else if (! grub_strcmp(param, "serial")) {
    } else if (! grub_strcmp(param, "terminal")) {
    } else if (! grub_strcmp(param, "title")) {
      current = new_entry(value, list);
    } else if (! grub_strcmp(param, "bootfs")) {
      char * lpath;
      if (! current) {
        grub_free(buf);
        grub_error (GRUB_ERR_INVALID_COMMAND, N_("illumos syntax error"));
        return NULL;
      }
      lpath = grub_strchr(value, '/');
      *lpath = 0;
      grub_strcpy(current->entry_info[POOL_LABEL], value);
      *lpath = '/';
      grub_strcpy(current->entry_info[DATA_SET], lpath);
    } else if (! grub_strcmp(param, "kernel$")) {
      char * ender;
      ender = grub_strchr(value, ' ');
      if (! ender)
        ender = grub_strchr(value, '\t');
      if (ender)
        *ender = 0;
      grub_strcpy(current->entry_info[KERNEL_PATH], value);
      if (ender) {
        ++ender;
        while ((*ender == ' ') || (*ender == '\t'))
          ++ender;
        grub_strcpy(current->entry_info[KERNEL_OPTIONS], ender);
      }
    } else if (! grub_strcmp(param, "module$")) {
      grub_strcpy(current->entry_info[BA_PATH], value);
    } else {
      grub_free(buf);
      continue;
    }
    grub_free(buf);
  }
  return list;
}

static grub_err_t
grub_solaris_legacy_cmd(struct grub_command *cmd,
      int argc, char **args)
{
```

```
128   int new_env, extractor;
129   extractor = (cmd->name[0] == 'e');
130   new_env = (cmd->name[extractor ? (sizeof ("extract_slegacy_entries_") - 1)
131             : (sizeof ("slegacy_") - 1)] == 'c');


134   if (argc != 1)
135     return grub_error (GRUB_ERR_BAD_ARGUMENT, N_("filename expected"));

137   grub_file_t file = grub_file_open(args[0]);

139   if (new_env)
140     grub_cls ();

142   if (new_env && !extractor)
143     grub_env_context_open ();
144   if (extractor)
145     grub_env_extractor_open (!new_env);

147   if (! file)
148     return grub_errno;

150   entries * list = parse_entries(file);

152   if (! list) {
153     return grub_error (GRUB_ERR_INVALID_COMMAND, N_("illumos syntax error"));
154   }
155
156   add_entries(list);

158   if (new_env)
159   {
160     grub_menu_t menu;
161     menu = grub_env_get_menu ();
162     if (menu && menu->size)
163     grub_show_menu (menu, 1, 0);
164     if (!extractor)
165       grub_env_context_close ();
166   }
167   if (extractor)
168     grub_env_extractor_close (!new_env);

170   clear_entries(list);

172   grub_file_close(file);
173   return 0;
174 }

176 static grub_command_t cmd_source, cmd_configfile;
177 static grub_command_t cmd_source_extract, cmd_configfile_extract;

179 GRUB_MOD_INIT(solaris_legacy)
180 {
181   cmd_source
182     = grub_register_command ("slegacy_source",
183           grub_solaris_legacy_cmd,
184           N_("FILE"),
185           /* TRANSLATORS: "legacy config" means
186         "config as used by grub-legacy".  */
187           N_("Parse legacy config in same context"));
188   cmd_configfile
189     = grub_register_command ("slegacy_configfile",
190           grub_solaris_legacy_cmd,
191           N_("FILE"),
192           N_("Parse legacy config in new context"));
193   cmd_source_extract
```



```
194     = grub_register_command ("extract_slegacy_entries_source",
195           grub_solaris_legacy_cmd,
196           N_("FILE"),
197           N_("Parse legacy config in same context taking only menu entries"));
198   cmd_configfile_extract
199     = grub_register_command ("extract_slegacy_entries_configfile",
200           grub_solaris_legacy_cmd,
201           N_("FILE"),
202           N_("Parse legacy config in new context taking only menu entries"));


205 }

207 GRUB_MOD_FINI(solaris_legacy)
208 {
209   grub_unregister_command (cmd_source);
210   grub_unregister_command (cmd_configfile);
211   grub_unregister_command (cmd_source_extract);
212   grub_unregister_command (cmd_configfile_extract);
213 }
214 #endif /* ! codereview */
```

```
**********************************************************
  107224 Fri Aug 31 05:08:52 2012
new/grub/grub-core/fs/zfs/zfs.c
grub patch
**********************************************************
```

```
/*
 *  GRUB  --  GRand Unified Bootloader
 *  Copyright (C) 1999,2000,2001,2002,2003,2004,2009,2010,2011  Free Software Fo
 *  Copyright 2010  Sun Microsystems, Inc.
 *  Copyright 2012  Daniil Lunev
#endif /* ! codereview */
 *
 *  GRUB is free software; you can redistribute it and/or modify
 *  it under the terms of the GNU General Public License as published by
 *  the Free Software Foundation; either version 3 of the License, or
 *  (at your option) any later version.
 *
 *  GRUB is distributed in the hope that it will be useful,
 *  but WITHOUT ANY WARRANTY; without even the implied warranty of
 *  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 *  GNU General Public License for more details.
 *
 *  You should have received a copy of the GNU General Public License
 *  along with GRUB.  If not, see <http://www.gnu.org/licenses/>.
 */
/*
 * The zfs plug-in routines for GRUB are:
 *
 * zfs_mount() - locates a valid uberblock of the root pool and reads
 *               in its MOS at the memory address MOS.
 *
 * zfs_open() - locates a plain file object by following the MOS
 *              and places its dnode at the memory address DNODE.
 *
 * zfs_read() - read in the data blocks pointed by the DNODE.
 *
 */

#include <grub/err.h>
#include <grub/file.h>
#include <grub/mm.h>
#include <grub/misc.h>
#include <grub/disk.h>
#include <grub/partition.h>
#include <grub/dl.h>
#include <grub/types.h>
#include <grub/zfs/zfs.h>
#include <grub/zfs/zio.h>
#include <grub/zfs/dnode.h>
#include <grub/zfs/uberblock_impl.h>
#include <grub/zfs/vdev_impl.h>
#include <grub/zfs/zio_checksum.h>
#include <grub/zfs/zap_impl.h>
#include <grub/zfs/zap_leaf.h>
#include <grub/zfs/zfs_znode.h>
#include <grub/zfs/dmu.h>
#include <grub/zfs/dmu_objset.h>
#include <grub/zfs/sa_impl.h>
#include <grub/zfs/dsl_dir.h>
#include <grub/zfs/dsl_dataset.h>
#include <grub/deflate.h>
#include <grub/crypto.h>
#include <grub/i18n.h>

GRUB_MOD_LICENSE ("GPLv3+");
```



```
#define ZPOOL_PROP_BOOTFS               "bootfs"

/*
 * For nvlist manipulation. (from nvpair.h)
 */
#define NV_ENCODE_NATIVE        0
#define NV_ENCODE_XDR           1
#define NV_BIG_ENDIAN           0
#define NV_LITTLE_ENDIAN        1
#define DATA_TYPE_UINT64        8
#define DATA_TYPE_STRING        9
#define DATA_TYPE_NVLIST        19
#define DATA_TYPE_NVLIST_ARRAY  20

#ifndef GRUB_UTIL
static grub_dl_t my_mod;
#endif

#define P2PHASE(x, align)               ((x) & ((align) - 1))

static inline grub_disk_addr_t
DVA_OFFSET_TO_PHYS_SECTOR (grub_disk_addr_t offset)
{
  return ((offset + VDEV_LABEL_START_SIZE) >> SPA_MINBLOCKSHIFT);
}

/*
 * FAT ZAP data structures
 */
#define ZFS_CRC64_POLY 0xC96C5795D7870F42ULL    /* ECMA-182, reflected form */
static inline grub_uint64_t
ZAP_HASH_IDX (grub_uint64_t hash, grub_uint64_t n)
{
  return (((n) == 0) ? 0 : ((hash) >> (64 - (n))));
}

#define CHAIN_END       0xffff  /* end of the chunk chain */

/*
 * The amount of space within the chunk available for the array is:
 * chunk size - space for type (1) - space for next pointer (2)
 */
#define ZAP_LEAF_ARRAY_BYTES (ZAP_LEAF_CHUNKSIZE - 3)

static inline int
ZAP_LEAF_HASH_SHIFT (int bs)
{
  return bs - 5;
}

static inline int
ZAP_LEAF_HASH_NUMENTRIES (int bs)
{
  return 1 << ZAP_LEAF_HASH_SHIFT(bs);
}

static inline grub_size_t
LEAF_HASH (int bs, grub_uint64_t h, zap_leaf_phys_t *l)
{
  return ((ZAP_LEAF_HASH_NUMENTRIES (bs)-1)
          & ((h) >> (64 - ZAP_LEAF_HASH_SHIFT (bs) - l->l_hdr.lh_prefix_len)));
}

/*
 * The amount of space available for chunks is:
 * block size shift - hash entry size (2) * number of hash
```

```
 * entries - header space (2*chunksize)
 */
static inline int
ZAP_LEAF_NUMCHUNKS (int bs)
{
  return (((1 << bs) - 2 * ZAP_LEAF_HASH_NUMENTRIES (bs)) /
          ZAP_LEAF_CHUNKSIZE - 2);
}

/*
 * The chunks start immediately after the hash table.  The end of the
 * hash table is at l_hash + HASH_NUMENTRIES, which we simply cast to a
 * chunk_t.
 */
static inline zap_leaf_chunk_t *
ZAP_LEAF_CHUNK (zap_leaf_phys_t *l, int bs, int idx)
{
  return &((zap_leaf_chunk_t *) (l->l_entries
                                 + (ZAP_LEAF_HASH_NUMENTRIES(bs) * 2)
                                 / sizeof (grub_properly_aligned_t)))[idx];
}

static inline struct zap_leaf_entry *
ZAP_LEAF_ENTRY(zap_leaf_phys_t *l, int bs, int idx)
{
  return &ZAP_LEAF_CHUNK(l, bs, idx)->l_entry;
}


/*
 * Decompression Entry - lzjb
 */

extern grub_err_t lzjb_decompress (void *, void *, grub_size_t, grub_size_t);

typedef grub_err_t zfs_decomp_func_t (void *s_start, void *d_start,
                                      grub_size_t s_len, grub_size_t d_len);
typedef struct decomp_entry
{
  const char *name;
  zfs_decomp_func_t *decomp_func;
} decomp_entry_t;

/*
 * Signature for checksum functions.
 */
typedef void zio_checksum_t(const void *data, grub_uint64_t size,
                            grub_zfs_endian_t endian, zio_cksum_t *zcp);

/*
 * Information about each checksum function.
 */
typedef struct zio_checksum_info {
        zio_checksum_t  *ci_func; /* checksum function for each byteorder */
        int             ci_correctable; /* number of correctable bits   */
        int             ci_eck;         /* uses zio embedded checksum? */
        const char              *ci_name;       /* descriptive name */
} zio_checksum_info_t;

typedef struct dnode_end
{
  dnode_phys_t dn;
  grub_zfs_endian_t endian;
} dnode_end_t;

struct grub_zfs_device_desc
```



```
{
  enum { DEVICE_LEAF, DEVICE_MIRROR, DEVICE_RAIDZ } type;
  grub_uint64_t id;
  grub_uint64_t guid;
  unsigned ashift;
  unsigned max_children_ashift;

  /* Valid only for non-leafs.  */
  unsigned n_children;
  struct grub_zfs_device_desc *children;

  /* Valid only for RAIDZ.  */
  unsigned nparity;

  /* Valid only for leaf devices.  */
  grub_device_t dev;
  grub_disk_addr_t vdev_phys_sector;
  uberblock_t current_uberblock;
  int original;
};

struct subvolume
{
  dnode_end_t mdn;
  grub_uint64_t obj;
  grub_uint64_t case_insensitive;
  grub_size_t nkeys;
  struct
  {
    grub_crypto_cipher_handle_t cipher;
    grub_uint64_t txg;
    grub_uint64_t algo;
  } *keyring;
};

static const char * feature_list[] = {
  "localhost:unknown_feature",
  NULL,
};

typedef enum zfs_feature_id {
  ZFS_FEATURE_UNKNOWN,
} zfs_feature_id_t;

struct enabled_feature_list {
  struct enabled_feature_list * next;
  zfs_feature_id_t id;
};

#endif /* ! codereview */
struct grub_zfs_data
{
  /* cache for a file block of the currently zfs_open()-ed file */
  char *file_buf;
  grub_uint64_t file_start;
  grub_uint64_t file_end;

  /* cache for a dnode block */
  dnode_phys_t *dnode_buf;
  dnode_phys_t *dnode_mdn;
  grub_uint64_t dnode_start;
  grub_uint64_t dnode_end;
  grub_zfs_endian_t dnode_endian;

  dnode_end_t mos;
  dnode_end_t dnode;
```

```
  struct subvolume subvol;

  struct grub_zfs_device_desc *devices_attached;
  unsigned n_devices_attached;
  unsigned n_devices_allocated;
  struct grub_zfs_device_desc *device_original;

  uberblock_t current_uberblock;

  struct enabled_feature_list * feature_list;

#endif /* ! codereview */
  int mounted;
  grub_uint64_t guid;
};

grub_err_t (*grub_zfs_decrypt) (grub_crypto_cipher_handle_t cipher,
                                grub_uint64_t algo,
                                void *nonce,
                                char *buf, grub_size_t size,
                                const grub_uint32_t *expected_mac,
                                grub_zfs_endian_t endian) = NULL;
grub_crypto_cipher_handle_t (*grub_zfs_load_key) (const struct grub_zfs_key *key
                                                  grub_size_t keysize,
                                                  grub_uint64_t salt,
                                                  grub_uint64_t algo) = NULL;

static grub_err_t
zlib_decompress (void *s, void *d,
                 grub_size_t slen, grub_size_t dlen)
{
  if (grub_zlib_decompress (s, slen, 0, d, dlen) < 0)
    return grub_errno;
  return GRUB_ERR_NONE;
}

static grub_err_t
zle_decompress (void *s, void *d,
                grub_size_t slen, grub_size_t dlen)
{
  grub_uint8_t *iptr, *optr;
  grub_size_t clen;
  for (iptr = s, optr = d; iptr < (grub_uint8_t *) s + slen
         && optr < (grub_uint8_t *) d + dlen;)
    {
      if (*iptr & 0x80)
        clen = ((*iptr) & 0x7f) + 0x41;
      else
        clen = ((*iptr) & 0x3f) + 1;
      if ((grub_ssize_t) clen > (grub_uint8_t *) d + dlen - optr)
        clen = (grub_uint8_t *) d + dlen - optr;
      if (*iptr & 0x40 || *iptr & 0x80)
        {
          grub_memset (optr, 0, clen);
          iptr++;
          optr += clen;
          continue;
        }
      if ((grub_ssize_t) clen > (grub_uint8_t *) s + slen - iptr - 1)
        clen = (grub_uint8_t *) s + slen - iptr - 1;
      grub_memcpy (optr, iptr + 1, clen);
      optr += clen;
      iptr += clen + 1;
    }
  if (optr < (grub_uint8_t *) d + dlen)
    grub_memset (optr, 0, (grub_uint8_t *) d + dlen - optr);
```



```
  return GRUB_ERR_NONE;
}

static decomp_entry_t decomp_table[ZIO_COMPRESS_FUNCTIONS] = {
  {"inherit", NULL},            /* ZIO_COMPRESS_INHERIT */
  {"on", lzjb_decompress},      /* ZIO_COMPRESS_ON */
  {"off", NULL},                /* ZIO_COMPRESS_OFF */
  {"lzjb", lzjb_decompress},    /* ZIO_COMPRESS_LZJB */
  {"empty", NULL},              /* ZIO_COMPRESS_EMPTY */
  {"gzip-1", zlib_decompress},  /* ZIO_COMPRESS_GZIP1 */
  {"gzip-2", zlib_decompress},  /* ZIO_COMPRESS_GZIP2 */
  {"gzip-3", zlib_decompress},  /* ZIO_COMPRESS_GZIP3 */
  {"gzip-4", zlib_decompress},  /* ZIO_COMPRESS_GZIP4 */
  {"gzip-5", zlib_decompress},  /* ZIO_COMPRESS_GZIP5 */
  {"gzip-6", zlib_decompress},  /* ZIO_COMPRESS_GZIP6 */
  {"gzip-7", zlib_decompress},  /* ZIO_COMPRESS_GZIP7 */
  {"gzip-8", zlib_decompress},  /* ZIO_COMPRESS_GZIP8 */
  {"gzip-9", zlib_decompress},  /* ZIO_COMPRESS_GZIP9 */
  {"zle", zle_decompress},      /* ZIO_COMPRESS_ZLE   */
};

static grub_err_t zio_read_data (blkptr_t * bp, grub_zfs_endian_t endian,
                                 void *buf, struct grub_zfs_data *data);

/*
 * Our own version of log2().  Same thing as highbit()-1.
 */
static int
zfs_log2 (grub_uint64_t num)
{
  int i = 0;

  while (num > 1)
    {
      i++;
      num = num >> 1;
    }

  return (i);
}

/* Checksum Functions */
static void
zio_checksum_off (const void *buf __attribute__ ((unused)),
                  grub_uint64_t size __attribute__ ((unused)),
                  grub_zfs_endian_t endian __attribute__ ((unused)),
                  zio_cksum_t * zcp)
{
  ZIO_SET_CHECKSUM (zcp, 0, 0, 0, 0);
}

/* Checksum Table and Values */
static zio_checksum_info_t zio_checksum_table[ZIO_CHECKSUM_FUNCTIONS] = {
  {NULL, 0, 0, "inherit"},
  {NULL, 0, 0, "on"},
  {zio_checksum_off, 0, 0, "off"},
  {zio_checksum_SHA256, 1, 1, "label"},
  {zio_checksum_SHA256, 1, 1, "gang_header"},
  {NULL, 0, 0, "zilog"},
  {fletcher_2, 0, 0, "fletcher2"},
  {fletcher_4, 1, 0, "fletcher4"},
  {zio_checksum_SHA256, 1, 0, "SHA256"},
  {NULL, 0, 0, "zilog2"},
  {zio_checksum_SHA256, 1, 0, "SHA256+MAC"},
};
```

```
/*
 * zio_checksum_verify: Provides support for checksum verification.
 *
 * Fletcher2, Fletcher4, and SHA256 are supported.
 *
 */
static grub_err_t
zio_checksum_verify (zio_cksum_t zc, grub_uint32_t checksum,
                     grub_zfs_endian_t endian,
                     char *buf, grub_size_t size)
{
  zio_eck_t *zec = (zio_eck_t *) (buf + size) - 1;
  zio_checksum_info_t *ci = &zio_checksum_table[checksum];
  zio_cksum_t actual_cksum, expected_cksum;

  if (checksum >= ZIO_CHECKSUM_FUNCTIONS || ci->ci_func == NULL)
    {
      grub_dprintf ("zfs", "unknown checksum function %d\n", checksum);
      return grub_error (GRUB_ERR_NOT_IMPLEMENTED_YET,
                 "unknown checksum function %d", checksum);
    }

  if (ci->ci_eck)
    {
      expected_cksum = zec->zec_cksum;
      zec->zec_cksum = zc;
      ci->ci_func (buf, size, endian, &actual_cksum);
      zec->zec_cksum = expected_cksum;
      zc = expected_cksum;
    }
  else
    ci->ci_func (buf, size, endian, &actual_cksum);

  if (grub_memcmp (&actual_cksum, &zc,
                   checksum != ZIO_CHECKSUM_SHA256_MAC ? 32 : 20) != 0)
    {
      grub_dprintf ("zfs", "checksum %s verification failed\n", ci->ci_name);
      grub_dprintf ("zfs", "actual checksum %016llx %016llx %016llx %016llx\n",
                    (unsigned long long) actual_cksum.zc_word[0],
                    (unsigned long long) actual_cksum.zc_word[1],
                    (unsigned long long) actual_cksum.zc_word[2],
                    (unsigned long long) actual_cksum.zc_word[3]);
      grub_dprintf ("zfs", "expected checksum %016llx %016llx %016llx %016llx\n"
                    (unsigned long long) zc.zc_word[0],
                    (unsigned long long) zc.zc_word[1],
                    (unsigned long long) zc.zc_word[2],
                    (unsigned long long) zc.zc_word[3]);
      return grub_error (GRUB_ERR_BAD_FS, N_("checksum verification failed"));
    }

  return GRUB_ERR_NONE;
}

/*
 * vdev_uberblock_compare takes two uberblock structures and returns an integer
 * indicating the more recent of the two.
 *      Return Value = 1 if ub2 is more recent
 *      Return Value = -1 if ub1 is more recent
 * The most recent uberblock is determined using its transaction number and
 * timestamp.  The uberblock with the highest transaction number is
 * considered "newer".  If the transaction numbers of the two blocks match, the
 * timestamps are compared to determine the "newer" of the two.
 */
static int
vdev_uberblock_compare (uberblock_t * ub1, uberblock_t * ub2)
{
```



```
  grub_zfs_endian_t ub1_endian, ub2_endian;
  if (grub_zfs_to_cpu64 (ub1->ub_magic, GRUB_ZFS_LITTLE_ENDIAN)
      == UBERBLOCK_MAGIC)
    ub1_endian = GRUB_ZFS_LITTLE_ENDIAN;
  else
    ub1_endian = GRUB_ZFS_BIG_ENDIAN;
  if (grub_zfs_to_cpu64 (ub2->ub_magic, GRUB_ZFS_LITTLE_ENDIAN)
      == UBERBLOCK_MAGIC)
    ub2_endian = GRUB_ZFS_LITTLE_ENDIAN;
  else
    ub2_endian = GRUB_ZFS_BIG_ENDIAN;

  if (grub_zfs_to_cpu64 (ub1->ub_txg, ub1_endian)
      < grub_zfs_to_cpu64 (ub2->ub_txg, ub2_endian))
    return (-1);
  if (grub_zfs_to_cpu64 (ub1->ub_txg, ub1_endian)
      > grub_zfs_to_cpu64 (ub2->ub_txg, ub2_endian))
    return (1);

  if (grub_zfs_to_cpu64 (ub1->ub_timestamp, ub1_endian)
      < grub_zfs_to_cpu64 (ub2->ub_timestamp, ub2_endian))
    return (-1);
  if (grub_zfs_to_cpu64 (ub1->ub_timestamp, ub1_endian)
      > grub_zfs_to_cpu64 (ub2->ub_timestamp, ub2_endian))
    return (1);

  return (0);
}

/*
 * Three pieces of information are needed to verify an uberblock: the magic
 * number, the version number, and the checksum.
 *
 * Currently Implemented: version number, magic number, checksum
 *
 */
static grub_err_t
uberblock_verify (uberblock_phys_t * ub, grub_uint64_t offset,
                  grub_size_t s)
{
  uberblock_t *uber = &ub->ubp_uberblock;
  grub_err_t err;
  grub_zfs_endian_t endian = GRUB_ZFS_UNKNOWN_ENDIAN;
  zio_cksum_t zc;

  if (grub_zfs_to_cpu64 (uber->ub_magic, GRUB_ZFS_LITTLE_ENDIAN)
      == UBERBLOCK_MAGIC
      && grub_zfs_to_cpu64 (uber->ub_version, GRUB_ZFS_LITTLE_ENDIAN) > 0)
      && grub_zfs_to_cpu64 (uber->ub_version, GRUB_ZFS_LITTLE_ENDIAN) > 0
      && grub_zfs_to_cpu64 (uber->ub_version, GRUB_ZFS_LITTLE_ENDIAN)
      <= SPA_VERSION)
    endian = GRUB_ZFS_LITTLE_ENDIAN;

  if (grub_zfs_to_cpu64 (uber->ub_magic, GRUB_ZFS_BIG_ENDIAN) == UBERBLOCK_MAGIC
      && grub_zfs_to_cpu64 (uber->ub_version, GRUB_ZFS_BIG_ENDIAN) > 0)
      && grub_zfs_to_cpu64 (uber->ub_version, GRUB_ZFS_BIG_ENDIAN) > 0
      && grub_zfs_to_cpu64 (uber->ub_version, GRUB_ZFS_BIG_ENDIAN)
      <= SPA_VERSION)
    endian = GRUB_ZFS_BIG_ENDIAN;

  if (endian == GRUB_ZFS_UNKNOWN_ENDIAN)
    return grub_error (GRUB_ERR_BAD_FS, "invalid uberblock magic");

  grub_memset (&zc, 0, sizeof (zc));

  zc.zc_word[0] = grub_cpu_to_zfs64 (offset, endian);
```

```
518   err = zio_checksum_verify (zc, ZIO_CHECKSUM_LABEL, endian,
519                              (char *) ub, s);

521   return err;
522 }
_____unchanged_portion_omitted_

782 /*
783  * Check the disk label information and retrieve needed vdev name-value pairs.
784  *
785  */
786 static grub_err_t
787 check_pool_label (struct grub_zfs_data *data,
788                   struct grub_zfs_device_desc *diskdesc,
789                   int *inserted)
790 {
791   grub_uint64_t pool_state, txg = 0;
792   char *nvlist;
793 #if 0
794   char *nv;
795 #endif
796   grub_uint64_t poolguid;
797   grub_uint64_t version;
798   int found;
799   grub_err_t err;

801   *inserted = 0;

803   err = zfs_fetch_nvlist (diskdesc, &nvlist);
804   if (err)
805     return err;

807   grub_dprintf ("zfs", "check 2 passed\n");

809   found = grub_zfs_nvlist_lookup_uint64 (nvlist, ZPOOL_CONFIG_POOL_STATE,
810                                          &pool_state);
811   if (! found)
812     {
813       grub_free (nvlist);
814       if (! grub_errno)
815         grub_error (GRUB_ERR_BAD_FS, ZPOOL_CONFIG_POOL_STATE " not found");
816       return grub_errno;
817     }
818   grub_dprintf ("zfs", "check 3 passed\n");

820   if (pool_state == POOL_STATE_DESTROYED)
821     {
822       grub_free (nvlist);
823       return grub_error (GRUB_ERR_BAD_FS, "zpool is marked as destroyed");
824     }
825   grub_dprintf ("zfs", "check 4 passed\n");

827   found = grub_zfs_nvlist_lookup_uint64 (nvlist, ZPOOL_CONFIG_POOL_TXG, &txg);
828   if (!found)
829     {
830       grub_free (nvlist);
831       if (! grub_errno)
832         grub_error (GRUB_ERR_BAD_FS, ZPOOL_CONFIG_POOL_TXG " not found");
833       return grub_errno;
834     }
835   grub_dprintf ("zfs", "check 6 passed\n");

837   /* not an active device */
838   if (txg == 0)
839     {
840       grub_free (nvlist);
```



```
841       return grub_error (GRUB_ERR_BAD_FS, "zpool isn't active");
842     }
843   grub_dprintf ("zfs", "check 7 passed\n");

845   found = grub_zfs_nvlist_lookup_uint64 (nvlist, ZPOOL_CONFIG_VERSION,
846                                          &version);
847   if (! found)
848     {
849       grub_free (nvlist);
850       if (! grub_errno)
851         grub_error (GRUB_ERR_BAD_FS, ZPOOL_CONFIG_VERSION " not found");
852       return grub_errno;
853     }
854   grub_dprintf ("zfs", "check 8 passed\n");

360   if (version > SPA_VERSION)
361     {
362       grub_free (nvlist);
363       return grub_error (GRUB_ERR_NOT_IMPLEMENTED_YET,
364                          "too new version %llu > %llu",
365                          (unsigned long long) version,
366                          (unsigned long long) SPA_VERSION);
367     }
855   grub_dprintf ("zfs", "check 9 passed\n");

857   found = grub_zfs_nvlist_lookup_uint64 (nvlist, ZPOOL_CONFIG_GUID,
858                                          &(diskdesc->guid));
859   if (! found)
860     {
861       grub_free (nvlist);
862       if (! grub_errno)
863         grub_error (GRUB_ERR_BAD_FS, ZPOOL_CONFIG_GUID " not found");
864       return grub_errno;
865     }

867   found = grub_zfs_nvlist_lookup_uint64 (nvlist, ZPOOL_CONFIG_POOL_GUID,
868                                          &poolguid);
869   if (! found)
870     {
871       grub_free (nvlist);
872       if (! grub_errno)
873         grub_error (GRUB_ERR_BAD_FS, ZPOOL_CONFIG_POOL_GUID " not found");
874       return grub_errno;
875     }

877   grub_dprintf ("zfs", "check 11 passed\n");

879   if (data->mounted && data->guid != poolguid)
880     return grub_error (GRUB_ERR_BAD_FS, "another zpool");
881   else
882     data->guid = poolguid;

884   {
885     char *nv;
886     nv = grub_zfs_nvlist_lookup_nvlist (nvlist, ZPOOL_CONFIG_VDEV_TREE);

888     if (!nv)
889       {
890         grub_free (nvlist);
891         return grub_error (GRUB_ERR_BAD_FS, "couldn't find vdev tree");
892       }
893     err = fill_vdev_info (data, nv, diskdesc, inserted);
894     if (err)
895       {
896         grub_free (nv);
897         grub_free (nvlist);
```

```
 898         return err;
 899       }
 900     grub_free (nv);
 901   }
 902   grub_dprintf ("zfs", "check 10 passed\n");

 904   grub_free (nvlist);

 906   return GRUB_ERR_NONE;
 907 }
_____unchanged_portion_omitted_

2680 #if 1
2193 #if 0
2681 /*
2682  * Get the default 'bootfs' property value from the rootpool.
2683  *
2684  */
2685 static grub_err_t
2686 get_default_bootfsobj (dnode_end_t * mosmdn, grub_uint64_t * obj,
2199 get_default_bootfsobj (dnode_phys_t * mosmdn, grub_uint64_t * obj,
2687                        struct grub_zfs_data *data)
2688 {
2689   grub_uint64_t objnum = 0;
2690   dnode_end_t dn;
2203   dnode_phys_t *dn;
2204   if (!dn)
2205     return grub_errno;

2692   if ((grub_errno = dnode_get (mosmdn, DMU_POOL_DIRECTORY_OBJECT,
2693                                DMU_OT_OBJECT_DIRECTORY, &dn, data)))
2208                                DMU_OT_OBJECT_DIRECTORY, dn, data)))
2694     {
2210       grub_free (dn);
2695       return (grub_errno);
2696     }

2697   /*
2698    * find the object number for 'pool_props', and get the dnode
2699    * of the 'pool_props'.
2700    */
2701   if (zap_lookup (&dn, DMU_POOL_PROPS, &objnum, data, 0))
2218   if (zap_lookup (dn, DMU_POOL_PROPS, &objnum, data))
2702     {
2220       grub_free (dn);
2703       return (GRUB_ERR_BAD_FS);
2704     }
2705   if ((grub_errno = dnode_get (mosmdn, objnum, DMU_OT_POOL_PROPS, &dn, data)))
2223   if ((grub_errno = dnode_get (mosmdn, objnum, DMU_OT_POOL_PROPS, dn, data)))
2706     {
2225       grub_free (dn);
2707       return (grub_errno);
2708     }
2709   if (zap_lookup (&dn, ZPOOL_PROP_BOOTFS, &objnum, data,0))
2228   if (zap_lookup (dn, ZPOOL_PROP_BOOTFS, &objnum, data))
2710     {
2230       grub_free (dn);
2711       return (GRUB_ERR_BAD_FS);
2712     }

2714   if (!objnum)
2715     {
2236       grub_free (dn);
2716       return (GRUB_ERR_BAD_FS);
2717     }
```



```
2719   *obj = objnum;
2720   return (0);
2721 }

2723 #endif /* ! codereview */
2724 #endif
2725 /*
2726  * Given a MOS metadnode, get the metadnode of a given filesystem name (fsname),
2727  * e.g. pool/rootfs, or a given object number (obj), e.g. the object number
2728  * of pool/rootfs.
2729  *
2730  * If no fsname and no obj are given, return the DSL_DIR metadnode.
2731  * If fsname is given, return its metadnode and its matching object number.
2732  * If only obj is given, return the metadnode for this object number.
2733  *
2734  */
2735 static grub_err_t
2736 get_filesystem_dnode (dnode_end_t * mosmdn, char *fsname,
2737                       dnode_end_t * mdn, struct grub_zfs_data *data)
2738 {
2739   grub_uint64_t objnum;
2740   grub_err_t err;

2742   grub_dprintf ("zfs", "endian = %d\n", mosmdn->endian);

2744   err = dnode_get (mosmdn, DMU_POOL_DIRECTORY_OBJECT,
2745                    DMU_OT_OBJECT_DIRECTORY, mdn, data);
2746   if (err)
2747     return err;

2749   grub_dprintf ("zfs", "alive\n");

2751   err = zap_lookup (mdn, DMU_POOL_ROOT_DATASET, &objnum, data, 0);
2752   if (err)
2753     return err;

2755   grub_dprintf ("zfs", "alive\n");

2757   err = dnode_get (mosmdn, objnum, DMU_OT_DSL_DIR, mdn, data);
2758   if (err)
2759     return err;

2761   grub_dprintf ("zfs", "alive\n");

2763   while (*fsname)
2764     {
2765       grub_uint64_t childobj;
2766       char *cname, ch;
2767
2768       while (*fsname == '/')
2769         fsname++;

2771       if (! *fsname || *fsname == '@')
2772         break;

2774       cname = fsname;
2775       while (*fsname && *fsname != '/')
2776         fsname++;
2777       ch = *fsname;
2778       *fsname = 0;

2780       childobj = grub_zfs_to_cpu64 ((((dsl_dir_phys_t *) DN_BONUS (&mdn->dn)))->
2781       err = dnode_get (mosmdn, childobj,
2782                        DMU_OT_DSL_DIR_CHILD_MAP, mdn, data);
2783       if (err)
2784         return err;
```

```
2786       err = zap_lookup (mdn, cname, &objnum, data, 0);
2787       if (err)
2788         return err;

2790       err = dnode_get (mosmdn, objnum, DMU_OT_DSL_DIR, mdn, data);
2791       if (err)
2792         return err;

2794       *fsname = ch;
2795     }
2796   return GRUB_ERR_NONE;
2797 }

2799 static grub_err_t
2800 make_mdn (dnode_end_t * mdn, struct grub_zfs_data *data)
2801 {
2802   void *osp;
2803   blkptr_t *bp;
2804   grub_size_t ospsize;
2805   grub_err_t err;

2807   grub_dprintf ("zfs", "endian = %d\n", mdn->endian);

2809   bp = &(((dsl_dataset_phys_t *) DN_BONUS (&mdn->dn))->ds_bp);
2810   err = zio_read (bp, mdn->endian, &osp, &ospsize, data);
2811   if (err)
2812     return err;
2813   if (ospsize < OBJSET_PHYS_SIZE_V14)
2814     {
2815       grub_free (osp);
2816       return grub_error (GRUB_ERR_BAD_FS, "too small osp");
2817     }

2819   mdn->endian = (grub_zfs_to_cpu64 (bp->blk_prop, mdn->endian)>>63) & 1;
2820   grub_memmove ((char *) &(mdn->dn),
2821                 (char *) &((objset_phys_t *) osp)->os_meta_dnode, DNODE_SIZE);
2822   grub_free (osp);
2823   return GRUB_ERR_NONE;
2824 }

2826 static grub_err_t
2827 dnode_get_fullpath (const char *fullpath, struct subvolume *subvol,
2828                     dnode_end_t * dn, int *isfs,
2829                     struct grub_zfs_data *data)
2830 {
2831   char *fsname, *snapname;
2832   const char *ptr_at, *filename;
2833   grub_uint64_t headobj;
2834   grub_uint64_t keychainobj;
2835   grub_uint64_t salt;
2836   grub_err_t err;
2837   int keyn = 0;

2839   auto int NESTED_FUNC_ATTR count_zap_keys (const void *name,
2840                                             grub_size_t namelen,
2841                                             const void *val_in,
2842                                             grub_size_t nelem,
2843                                             grub_size_t elemsize);
2844   int NESTED_FUNC_ATTR count_zap_keys (const void *name __attribute__ ((unused))
2845                                        grub_size_t namelen __attribute__ ((unuse
2846                                        const void *val_in __attribute__ ((unused
2847                                        grub_size_t nelem __attribute__ ((unused)
2848                                        grub_size_t elemsize __attribute__ ((unus
2849   {
2850     subvol->nkeys++;
```



```
2851     return 0;
2852   }

2854   auto int NESTED_FUNC_ATTR load_zap_key (const void *name,
2855                                           grub_size_t namelen,
2856                                           const void *val_in,
2857                                           grub_size_t nelem,
2858                                           grub_size_t elemsize);
2859   int NESTED_FUNC_ATTR load_zap_key (const void *name,
2860                                      grub_size_t namelen,
2861                                      const void *val_in,
2862                                      grub_size_t nelem,
2863                                      grub_size_t elemsize)
2864   {
2865     if (namelen != 1)
2866       {
2867         grub_dprintf ("zfs", "Unexpected key index size %" PRIuGRUB_SIZE "\n",
2868                       namelen);
2869         return 0;
2870       }

2872     if (elemsize != 1)
2873       {
2874         grub_dprintf ("zfs", "Unexpected key element size %" PRIuGRUB_SIZE "\n",
2875                       elemsize);
2876         return 0;
2877       }

2879     subvol->keyring[keyn].txg = grub_be_to_cpu64 (*(grub_uint64_t *) name);
2880     subvol->keyring[keyn].algo = grub_le_to_cpu64 (*(grub_uint64_t *) val_in);
2881     subvol->keyring[keyn].cipher = grub_zfs_load_key (val_in, nelem, salt,
2882                                                       subvol->keyring[keyn].algo
2883     keyn++;
2884     return 0;
2885   }

2887   ptr_at = grub_strchr (fullpath, '@');
2888   if (! ptr_at)
2889     {
2890       *isfs = 1;
2891       filename = 0;
2892       snapname = 0;
2893       fsname = grub_strdup (fullpath);
2894     }
2895   else
2896     {
2897       const char *ptr_slash = grub_strchr (ptr_at, '/');

2899       *isfs = 0;
2900       fsname = grub_malloc (ptr_at - fullpath + 1);
2901       if (!fsname)
2902         return grub_errno;
2903       grub_memcpy (fsname, fullpath, ptr_at - fullpath);
2904       fsname[ptr_at - fullpath] = 0;
2905       if (ptr_at[1] && ptr_at[1] != '/')
2906         {
2907           snapname = grub_malloc (ptr_slash - ptr_at);
2908           if (!snapname)
2909             {
2910               grub_free (fsname);
2911               return grub_errno;
2912             }
2913           grub_memcpy (snapname, ptr_at + 1, ptr_slash - ptr_at - 1);
2914           snapname[ptr_slash - ptr_at - 1] = 0;
2915         }
2916       else
```

```
2917          snapname = 0;
2918        if (ptr_slash)
2919          filename = ptr_slash;
2920        else
2921          filename = "/";
2922        grub_dprintf ("zfs", "fsname = '%s' snapname='%s' filename = '%s'\n",
2923                      fsname, snapname, filename);
2924      }
2925    grub_dprintf ("zfs", "alive\n");
2926    err = get_filesystem_dnode (&(data->mos), fsname, dn, data);
2927    if (err)
2928      {
2929        grub_free (fsname);
2930        grub_free (snapname);
2931        return err;
2932      }

2934    grub_dprintf ("zfs", "alive\n");

2936    headobj = grub_zfs_to_cpu64 (((dsl_dir_phys_t *) DN_BONUS (&dn->dn))->dd_head_

2938    grub_dprintf ("zfs", "endian = %d\n", subvol->mdn.endian);

2940    err = dnode_get (&(data->mos), headobj, DMU_OT_DSL_DATASET, &subvol->mdn,
2941                     data);
2942    if (err)
2943      {
2944        grub_free (fsname);
2945        grub_free (snapname);
2946        return err;
2947      }
2948    grub_dprintf ("zfs", "endian = %d\n", subvol->mdn.endian);

2950    keychainobj = grub_zfs_to_cpu64 (((dsl_dir_phys_t *) DN_BONUS (&dn->dn))->keyc
2951    if (grub_zfs_load_key && keychainobj)
2952      {
2953        dnode_end_t keychain_dn, props_dn;
2954        grub_uint64_t propsobj;
2955        propsobj = grub_zfs_to_cpu64 (((dsl_dir_phys_t *) DN_BONUS (&dn->dn))->dd_

2957        err = dnode_get (&(data->mos), propsobj, DMU_OT_DSL_PROPS,
2958                         &props_dn, data);
2959        if (err)
2960          {
2961            grub_free (fsname);
2962            grub_free (snapname);
2963            return err;
2964          }

2966        err = zap_lookup (&props_dn, "salt", &salt, data, 0);
2967        if (err == GRUB_ERR_FILE_NOT_FOUND)
2968          {
2969            err = 0;
2970            grub_errno = 0;
2971            salt = 0;
2972          }
2973        if (err)
2974          {
2975            grub_dprintf ("zfs", "failed here\n");
2976            return err;
2977          }

2979        err = dnode_get (&(data->mos), keychainobj, DMU_OT_DSL_KEYCHAIN,
2980                         &keychain_dn, data);
2981        if (err)
2982          {
```



```
2983            grub_free (fsname);
2984            grub_free (snapname);
2985            return err;
2986          }
2987        subvol->nkeys = 0;
2988        zap_iterate (&keychain_dn, 8, count_zap_keys, data);
2989        subvol->keyring = grub_zalloc (subvol->nkeys * sizeof (subvol->keyring[0])
2990        if (!subvol->keyring)
2991          {
2992            grub_free (fsname);
2993            grub_free (snapname);
2994            return err;
2995          }
2996        zap_iterate (&keychain_dn, 8, load_zap_key, data);
2997      }

2999    if (snapname)
3000      {
3001        grub_uint64_t snapobj;

3003        snapobj = grub_zfs_to_cpu64 (((dsl_dataset_phys_t *) DN_BONUS (&subvol->md

3005        err = dnode_get (&(data->mos), snapobj,
3006                         DMU_OT_DSL_DS_SNAP_MAP, &subvol->mdn, data);
3007        if (!err)
3008          err = zap_lookup (&subvol->mdn, snapname, &headobj, data, 0);
3009        if (!err)
3010          err = dnode_get (&(data->mos), headobj, DMU_OT_DSL_DATASET,
3011                           &subvol->mdn, data);
3012        if (err)
3013          {
3014            grub_free (fsname);
3015            grub_free (snapname);
3016            return err;
3017          }
3018      }

3020    subvol->obj = headobj;

3022    make_mdn (&subvol->mdn, data);
3023
3024    grub_dprintf ("zfs", "endian = %d\n", subvol->mdn.endian);

3026    if (*isfs)
3027      {
3028        grub_free (fsname);
3029        grub_free (snapname);
3030        return GRUB_ERR_NONE;
3031      }
3032    err = dnode_get_path (subvol, filename, dn, data);
3033    grub_free (fsname);
3034    grub_free (snapname);
3035    return err;
3036  }

3038  /*
3039   * For a given XDR packed nvlist, verify the first 4 bytes and move on.
3040   *
3041   * An XDR packed nvlist is encoded as (comments from nvs_xdr_create) :
3042   *
3043   *      encoding method/host endian     (4 bytes)
3044   *      nvl_version                     (4 bytes)
3045   *      nvl_nvflag                      (4 bytes)
3046   *      encoded nvpairs:
3047   *              encoded size of the nvpair      (4 bytes)
3048   *              decoded size of the nvpair      (4 bytes)
```

```
3049  *              name string size                (4 bytes)
3050  *              name string data                (sizeof(NV_ALIGN4(string))
3051  *              data type                       (4 bytes)
3052  *              # of elements in the nvpair     (4 bytes)
3053  *              data
3054  *      2 zero's for the last nvpair
3055  *              (end of the entire list)        (8 bytes)
3056  *
3057  */

3059 static int
3060 nvlist_find_value (const char *nvlist_in, const char *name,
3061                    int valtype, char **val,
3062                    grub_size_t *size_out, grub_size_t *nelm_out)
3063 {
3064   int name_len, type, encode_size;
3065   const char *nvpair, *nvp_name, *nvlist = nvlist_in;

3067   /* Verify if the 1st and 2nd byte in the nvlist are valid. */
3068   /* NOTE: independently of what endianness header announces all
3069      subsequent values are big-endian.  */
3070   if (nvlist[0] != NV_ENCODE_XDR || (nvlist[1] != NV_LITTLE_ENDIAN
3071                                      && nvlist[1] != NV_BIG_ENDIAN))
3072     {
3073       grub_dprintf ("zfs", "incorrect nvlist header\n");
3074       grub_error (GRUB_ERR_BAD_FS, "incorrect nvlist");
3075       return 0;
3076     }

3078   /* skip the header, nvl_version, and nvl_nvflag */
3079   nvlist = nvlist + 4 * 3;
3080   /*
3081    * Loop thru the nvpair list
3082    * The XDR representation of an integer is in big-endian byte order.
3083    */
3084   while ((encode_size = grub_be_to_cpu32 (grub_get_unaligned32 (nvlist))))
3085     {
3086       int nelm;

3088       if (nvlist + 4 * 4 >= nvlist_in + VDEV_PHYS_SIZE)
3089         {
3090           grub_dprintf ("zfs", "nvlist overflow\n");
3091           grub_error (GRUB_ERR_BAD_FS, "incorrect nvlist");
3092           return 0;
3093         }

3095       nvpair = nvlist + 4 * 2;  /* skip the encode/decode size */

3097       name_len = grub_be_to_cpu32 (grub_get_unaligned32 (nvpair));
3098       nvpair += 4;

3100       nvp_name = nvpair;
3101       nvpair = nvpair + ((name_len + 3) & ~3);  /* align */

3103       if (nvpair + 8 >= nvlist_in + VDEV_PHYS_SIZE
3104           || encode_size < 0
3105           || nvpair + 8 + encode_size > nvlist_in + VDEV_PHYS_SIZE)
3106         {
3107           grub_dprintf ("zfs", "nvlist overflow\n");
3108           grub_error (GRUB_ERR_BAD_FS, "incorrect nvlist");
3109           return 0;
3110         }

3112       type = grub_be_to_cpu32 (grub_get_unaligned32 (nvpair));
3113       nvpair += 4;
```



```
3115       nelm = grub_be_to_cpu32 (grub_get_unaligned32 (nvpair));
3116       if (nelm < 1)
3117         {
3118           grub_error (GRUB_ERR_BAD_FS, "empty nvpair");
3119           return 0;
3120         }

3122       nvpair += 4;

3124       if ((grub_strncmp (nvp_name, name, name_len) == 0) && type == valtype)
3125         {
3126           *val = (char *) nvpair;
3127           *size_out = encode_size;
3128           if (nelm_out)
3129             *nelm_out = nelm;
3130           return 1;
3131         }

3133       nvlist += encode_size;    /* goto the next nvpair */
3134     }
3135   return 0;
3136 }

3138 int
3139 grub_zfs_nvlist_lookup_uint64 (const char *nvlist, const char *name,
3140                                grub_uint64_t * out)
3141 {
3142   char *nvpair;
3143   grub_size_t size;
3144   int found;

3146   found = nvlist_find_value (nvlist, name, DATA_TYPE_UINT64, &nvpair, &size, 0);
3147   if (!found)
3148     return 0;
3149   if (size < sizeof (grub_uint64_t))
3150     {
3151       grub_error (GRUB_ERR_BAD_FS, "invalid uint64");
3152       return 0;
3153     }

3155   *out = grub_be_to_cpu64 (grub_get_unaligned64 (nvpair));
3156   return 1;
3157 }

3159 char *
3160 grub_zfs_nvlist_lookup_string (const char *nvlist, const char *name)
3161 {
3162   char *nvpair;
3163   char *ret;
3164   grub_size_t slen;
3165   grub_size_t size;
3166   int found;

3168   found = nvlist_find_value (nvlist, name, DATA_TYPE_STRING, &nvpair, &size, 0);
3169   if (!found)
3170     return 0;
3171   if (size < 4)
3172     {
3173       grub_error (GRUB_ERR_BAD_FS, "invalid string");
3174       return 0;
3175     }
3176   slen = grub_be_to_cpu32 (grub_get_unaligned32 (nvpair));
3177   if (slen > size - 4)
3178     slen = size - 4;
3179   ret = grub_malloc (slen + 1);
3180   if (!ret)
```

```
3181     return 0;
3182   grub_memcpy (ret, nvpair + 4, slen);
3183   ret[slen] = 0;
3184   return ret;
3185 }

3187 char *
3188 grub_zfs_nvlist_lookup_nvlist (const char *nvlist, const char *name)
3189 {
3190   char *nvpair;
3191   char *ret;
3192   grub_size_t size;
3193   int found;

3195   found = nvlist_find_value (nvlist, name, DATA_TYPE_NVLIST, &nvpair,
3196                              &size, 0);
3197   if (!found)
3198     return 0;
3199   ret = grub_zalloc (size + 3 * sizeof (grub_uint32_t));
3200   if (!ret)
3201     return 0;
3202   grub_memcpy (ret, nvlist, sizeof (grub_uint32_t));

3204   grub_memcpy (ret + sizeof (grub_uint32_t), nvpair, size);
3205   return ret;
3206 }

3208 int
3209 grub_zfs_nvlist_lookup_nvlist_array_get_nelm (const char *nvlist,
3210                                               const char *name)
3211 {
3212   char *nvpair;
3213   grub_size_t nelm, size;
3214   int found;

3216   found = nvlist_find_value (nvlist, name, DATA_TYPE_NVLIST_ARRAY, &nvpair,
3217                              &size, &nelm);
3218   if (! found)
3219     return -1;
3220   return nelm;
3221 }

3223 static int
3224 get_nvlist_size (const char *beg, const char *limit)
3225 {
3226   const char *ptr;
3227   grub_uint32_t encode_size;
3228
3229   ptr = beg + 8;

3231   while (ptr < limit
3232          && (encode_size = grub_be_to_cpu32 (grub_get_unaligned32 (ptr))))
3233     ptr += encode_size; /* goto the next nvpair */
3234   ptr += 8;
3235   return (ptr > limit) ? -1 : (ptr - beg);
3236 }

3238 char *
3239 grub_zfs_nvlist_lookup_nvlist_array (const char *nvlist, const char *name,
3240                                      grub_size_t index)
3241 {
3242   char *nvpair, *nvpairptr;
3243   int found;
3244   char *ret;
3245   grub_size_t size;
3246   unsigned i;
```



```
3247   grub_size_t nelm;
3248   int elemsize = 0;

3250   found = nvlist_find_value (nvlist, name, DATA_TYPE_NVLIST_ARRAY, &nvpair,
3251                              &size, &nelm);
3252   if (!found)
3253     return 0;
3254   if (index >= nelm)
3255     {
3256       grub_error (GRUB_ERR_OUT_OF_RANGE, "trying to lookup past nvlist array");
3257       return 0;
3258     }

3260   nvpairptr = nvpair;

3262   for (i = 0; i < index; i++)
3263     {
3264       int r;
3265       r = get_nvlist_size (nvpairptr, nvpair + size);

3267       if (r < 0)
3268         {
3269           grub_error (GRUB_ERR_BAD_FS, "incorrect nvlist array");
3270           return NULL;
3271         }
3272       nvpairptr += r;
3273     }

3275   elemsize = get_nvlist_size (nvpairptr, nvpair + size);

3277   if (elemsize < 0)
3278     {
3279       grub_error (GRUB_ERR_BAD_FS, "incorrect nvlist array");
3280       return 0;
3281     }

3283   ret = grub_zalloc (elemsize + sizeof (grub_uint32_t));
3284   if (!ret)
3285     return 0;
3286   grub_memcpy (ret, nvlist, sizeof (grub_uint32_t));

3288   grub_memcpy (ret + sizeof (grub_uint32_t), nvpairptr, elemsize);
3289   return ret;
3290 }

3292 static void
3293 unmount_device (struct grub_zfs_device_desc *desc)
3294 {
3295   unsigned i;
3296   switch (desc->type)
3297     {
3298     case DEVICE_LEAF:
3299       if (!desc->original && desc->dev)
3300         grub_device_close (desc->dev);
3301       return;
3302     case DEVICE_RAIDZ:
3303     case DEVICE_MIRROR:
3304       for (i = 0; i < desc->n_children; i++)
3305         unmount_device (&desc->children[i]);
3306       grub_free (desc->children);
3307       return;
3308     }
3309 }

3311 static void
3312 zfs_unmount (struct grub_zfs_data *data)
```

```
3313 {
3314   unsigned i;
3315   for (i = 0; i < data->n_devices_attached; i++)
3316     unmount_device (&data->devices_attached[i]);
3317   grub_free (data->devices_attached);
3318   grub_free (data->dnode_buf);
3319   grub_free (data->dnode_mdn);
3320   grub_free (data->file_buf);
3321   for (i = 0; i < data->subvol.nkeys; i++)
3322     grub_crypto_cipher_close (data->subvol.keyring[i].cipher);
3323   grub_free (data->subvol.keyring);
3324   while (data->feature_list) {
3325     struct enabled_feature_list * tmp = data->feature_list;
3326     data->feature_list = data->feature_list->next;
3327     grub_free(tmp);
3328   }
3329 #endif /* ! codereview */
3330   grub_free (data);
3331 }

3333 static int
3334 add_feature (struct grub_zfs_data * data, zfs_feature_id_t id)
3335 {
3336   struct enabled_feature_list * list = data->feature_list;
3337
3338   while (list->next) {
3339     if (list->next->id == id)
3340       return id;
3341     list = list->next;
3342   }
3343
3344   list->next = (struct enabled_feature_list *) grub_zalloc (sizeof (struct enabl
3345
3346   if (! list->next)
3347     return (-1);

3349   list->next->id = id;
3350   list->next->next = NULL;
3351
3352   return 0;
3353 }

3355 static zfs_feature_id_t
3356 check_feature (const char * feature)
3357 {
3358   int i = 0;
3359   for ( ; i < ZFS_FEATURE_UNKNOWN; ++i)
3360     if (! grub_strcmp (feature, feature_list[i]))
3361           return i;

3363   return ZFS_FEATURE_UNKNOWN;
3364 }

3366 /*
3367  * zfs_get_features_list() get feature list and check compatability
3368  */
3369 static grub_err_t
3370 zfs_get_features_list (struct grub_zfs_data * data)
3371 {
3372   dnode_end_t dn;
3373   dnode_end_t * mos = &(data->mos);
3374   grub_err_t err;
3375   grub_uint64_t objnum;
3376   int ret;
3377
3378   int NESTED_FUNC_ATTR feature_hook (const char * cname, grub_uint64_t val)
```



```
3379   {
3380     grub_err_t iret = 0;

3382     zfs_feature_id_t id;

3384     if (! (*cname))
3385       goto out;

3387     // retrieve feature number
3388     id = check_feature (cname);

3390     // if grub doesn't support such feature, return error
3391     if (id == ZFS_FEATURE_UNKNOWN) {
3392       if (val == 0) {
3393         grub_error(GRUB_ERR_BAD_FS,
3394               "Unsupported feature %s was enabled,"
3395               " but haven't been activated yet. You will not be able to boot"
3396               " if this feature are activated.", cname);
3397       } else {
3398         grub_error(GRUB_ERR_BAD_FS,
3399             "Unsupported feature %s is activated. Booting failed",cname);
3400         iret = 1;
3401       }
3402       goto out;
3403     }
3404
3405     // add feature to list
3406     iret = add_feature(data, id);
3407
3408   out:
3409     return iret;
3410   }
3411
3412   // get object directory
3413   err = dnode_get (mos, DMU_POOL_DIRECTORY_OBJECT,
3414             DMU_OT_OBJECT_DIRECTORY, &dn, data);
3415   if (err)
3416     return err;

3418   // retrieve pool properties object numer
3419   err = zap_lookup (&dn, DMU_POOL_FEATURES_FOR_READ, &objnum, data, 0);
3420   if (err)
3421     return err;
3422
3423   // get "features for read" zap dnode
3424   err = dnode_get (mos, objnum, DMU_OTN_ZAP_METADATA, &dn, data);
3425   if (err)
3426     return err;
3427
3428   // itterate zap to fetch feature list
3429   ret = zap_iterate_u64 (&dn, feature_hook, data);
3430   if (ret)
3431     return grub_error(GRUB_ERR_BAD_FS, "There are enabled unsupported features")
3432
3433   return GRUB_ERR_NONE;
3434 }

3436 #endif /* ! codereview */
3437 /*
3438  * zfs_mount() locates a valid uberblock of the root pool and read in its MOS
3439  * to the memory address MOS.
3440  *
3441  */
3442 static struct grub_zfs_data *
3443 zfs_mount (grub_device_t dev)
3444 {
```

```
3445   struct grub_zfs_data *data = 0;
3446   grub_err_t err;
3447   void *osp = 0;
3448   grub_size_t ospsize;
3449   grub_zfs_endian_t ub_endian = GRUB_ZFS_UNKNOWN_ENDIAN;
3450   uberblock_t *ub;
3451   int inserted;

3453   if (! dev->disk)
3454     {
3455       grub_error (GRUB_ERR_BAD_DEVICE, "not a disk");
3456       return 0;
3457     }

3459   data = grub_zalloc (sizeof (*data));
3460   if (!data)
3461     return 0;
3462 #if 0
3463   /* if it's our first time here, zero the best uberblock out */
3464   if (data->best_drive == 0 && data->best_part == 0 && find_best_root)
3465     grub_memset (&current_uberblock, 0, sizeof (uberblock_t));
3466 #endif

3468   data->feature_list = NULL;
3469 #endif /* ! codereview */
3470   data->n_devices_allocated = 16;
3471   data->devices_attached = grub_malloc (sizeof (data->devices_attached[0])
3472                                         * data->n_devices_allocated);
3473   data->n_devices_attached = 0;
3474   err = scan_disk (dev, data, 1, &inserted);
3475   if (err)
3476     {
3477       zfs_unmount (data);
3478       return NULL;
3479     }

3481   ub = &(data->current_uberblock);
3482   ub_endian = (grub_zfs_to_cpu64 (ub->ub_magic,
3483                                   GRUB_ZFS_LITTLE_ENDIAN) == UBERBLOCK_MAGIC
3484                ? GRUB_ZFS_LITTLE_ENDIAN : GRUB_ZFS_BIG_ENDIAN);

3486   err = zio_read (&ub->ub_rootbp, ub_endian,
3487                   &osp, &ospsize, data);
3488   if (err)
3489     {
3490       zfs_unmount (data);
3491       return NULL;
3492     }

3494   if (ospsize < OBJSET_PHYS_SIZE_V14)
3495     {
3496       grub_error (GRUB_ERR_BAD_FS, "OSP too small");
3497       grub_free (osp);
3498       zfs_unmount (data);
3499       return NULL;
3500     }

3502   /* Got the MOS. Save it at the memory addr MOS. */
3503   grub_memmove (&(data->mos.dn), &((objset_phys_t *) osp)->os_meta_dnode,
3504                 DNODE_SIZE);
3505   data->mos.endian = (grub_zfs_to_cpu64 (ub->ub_rootbp.blk_prop,
3506                                          ub_endian) >> 63) & 1;
3507   grub_free (osp);
3508
3509   data->mounted = 1;
```



```
3511   if (grub_zfs_to_cpu64(ub->ub_version, ub_endian) == SPA_FEATURE_VERSION) {
3512     data->feature_list = (struct enabled_feature_list*) grub_zalloc (
3513                                                    sizeof (struct enabled_feature
3514     data->feature_list->next = NULL;
3515     err = zfs_get_features_list(data);
3516
3517     if (err) {
3518       data->mounted = 0;
3519       zfs_unmount (data);
3520       return NULL;
3521     }
3522   }

3524 #endif /* ! codereview */
3525   return data;
3526 }

3528 static grub_err_t
3529 find_default_dataset_path(char * path, grub_uint64_t * mdnobj, struct grub_zfs_d
3530 {
3531   grub_uint64_t obj, pobj, zobj;
3532   grub_err_t err;
3533   dnode_end_t mdn, *mosmdn;
3534   char buf[512];

3536   int NESTED_FUNC_ATTR hook (const char * name, grub_uint64_t val) {
3537     if (val == obj) {
3538       grub_strcpy(buf, name);
3539       return 1;
3540     }
3541     return 0;
3542   }

3544   obj = *mdnobj;
3545   mosmdn = &(data->mos);
3546   buf[0] = '\0';
3547   path[0] = '\0';

3549   // get object's data dir
3550   err = dnode_get(mosmdn, obj, DMU_OT_DSL_DATASET, &mdn, data);
3551   if (err)
3552     return err;
3553   obj = grub_zfs_to_cpu64((((dsl_dataset_phys_t*)DN_BONUS (&mdn.dn)))->ds_dir_ob
3554
3555   for (;;) {
3556     // get object dnode
3557     err = dnode_get(mosmdn, obj, DMU_OT_DSL_DIR, &mdn, data);
3558     if (err)
3559       return err;

3561     // find out its parent's objnum
3562     pobj = grub_zfs_to_cpu64((((dsl_dir_phys_t*)DN_BONUS (&mdn.dn)))->dd_parent_
3563     if (obj == pobj)
3564       break;

3566     // if pobj is 0 then we have reached top dataset
3567     if (! pobj)
3568       break;

3570     // get object's parent dnode
3571     err = dnode_get(mosmdn, pobj, DMU_OT_DSL_DIR, &mdn, data);
3572     if (err)
3573       return err;

3575     // find out parent's zap objnum
3576     zobj = grub_zfs_to_cpu64((((dsl_dir_phys_t*)DN_BONUS (&mdn.dn)))->dd_child_d
```

```
3578     // get zap's dnode
3579     err = dnode_get(mosmdn, zobj, DMU_OT_DSL_DIR_CHILD_MAP, &mdn, data);
3580     if (err)
3581       return err;

3583     // lookup zap to get name
3584     err = zap_iterate_u64(&mdn, hook, data);
3585     if (err == 0)
3586       return err;

3588     // pobj becomes obj now
3589     obj = pobj;

3591     // append new intermediate dnode to path
3592     grub_strcat(buf, path);
3593     grub_strcpy(path, "/");
3594     grub_strcat(path, buf);
3595   }
3596
3597   return GRUB_ERR_NONE;
3598 }

3600 grub_err_t
3601 get_default_bootfs_obj(grub_device_t dev, char * path, grub_uint64_t * mdnobj)
3602 {
3603   struct grub_zfs_data * data;
3604   grub_err_t err = 0;
3605   data = zfs_mount(dev);
3606   if (data) {
3607     err = get_default_bootfsobj(&(data->mos), mdnobj, data);
3608     if (err)
3609       return err;
3610     err = find_default_dataset_path(path, mdnobj, data);
3611     zfs_unmount(data);
3612     return err;
3613   } else {
3614     return grub_errno;
3615   }
3616   return 0;
3617 }

3619 #endif /* ! codereview */
3620 grub_err_t
3621 grub_zfs_fetch_nvlist (grub_device_t dev, char **nvlist)
3622 {
3623   struct grub_zfs_data *zfs;
3624   grub_err_t err;

3626   zfs = zfs_mount (dev);
3627   if (!zfs)
3628     return grub_errno;
3629   err = zfs_fetch_nvlist (zfs->device_original, nvlist);
3630   zfs_unmount (zfs);
3631   return err;
3632 }

3634 static grub_err_t
3635 zfs_label (grub_device_t device, char **label)
3636 {
3637   char *nvlist;
3638   grub_err_t err;
3639   struct grub_zfs_data *data;

3641   data = zfs_mount (device);
3642   if (! data)
```



```
3643     return grub_errno;

3645   err = zfs_fetch_nvlist (data->device_original, &nvlist);
3646   if (err)
3647     {
3648       zfs_unmount (data);
3649       return err;
3650     }

3652   *label = grub_zfs_nvlist_lookup_string (nvlist, ZPOOL_CONFIG_POOL_NAME);
3653   grub_free (nvlist);
3654   zfs_unmount (data);
3655   return grub_errno;
3656 }

3658 static grub_err_t
3659 zfs_uuid (grub_device_t device, char **uuid)
3660 {
3661   struct grub_zfs_data *data;

3663   *uuid = 0;

3665   data = zfs_mount (device);
3666   if (! data)
3667     return grub_errno;

3669   *uuid = grub_xasprintf ("%016llx", (long long unsigned) data->guid);
3670   zfs_unmount (data);
3671   if (! *uuid)
3672     return grub_errno;
3673   return GRUB_ERR_NONE;
3674 }

3676 static grub_err_t
3677 zfs_mtime (grub_device_t device, grub_int32_t *mt)
3678 {
3679   struct grub_zfs_data *data;
3680   grub_zfs_endian_t ub_endian = GRUB_ZFS_UNKNOWN_ENDIAN;
3681   uberblock_t *ub;

3683   *mt = 0;

3685   data = zfs_mount (device);
3686   if (! data)
3687     return grub_errno;

3689   ub = &(data->current_uberblock);
3690   ub_endian = (grub_zfs_to_cpu64 (ub->ub_magic,
3691                                   GRUB_ZFS_LITTLE_ENDIAN) == UBERBLOCK_MAGIC
3692                ? GRUB_ZFS_LITTLE_ENDIAN : GRUB_ZFS_BIG_ENDIAN);

3694   *mt = grub_zfs_to_cpu64 (ub->ub_timestamp, ub_endian);
3695   zfs_unmount (data);
3696   return GRUB_ERR_NONE;
3697 }

3699 /*
3700  * zfs_open() locates a file in the rootpool by following the
3701  * MOS and places the dnode of the file in the memory address DNODE.
3702  */
3703 static grub_err_t
3704 grub_zfs_open (struct grub_file *file, const char *fsfilename)
3705 {
3706   struct grub_zfs_data *data;
3707   grub_err_t err;
3708   int isfs;
```

```
3710   data = zfs_mount (file->device);
3711   if (! data)
3712     return grub_errno;

3714   err = dnode_get_fullpath (fsfilename, &(data->subvol),
3715                             &(data->dnode), &isfs, data);
3716   if (err)
3717     {
3718       zfs_unmount (data);
3719       return err;
3720     }

3722   if (isfs)
3723     {
3724       zfs_unmount (data);
3725       return grub_error (GRUB_ERR_BAD_FILE_TYPE, N_("missing ‘%c’ symbol"), ’@’)
3726     }

3728   /* We found the dnode for this file. Verify if it is a plain file. */
3729   if (data->dnode.dn.dn_type != DMU_OT_PLAIN_FILE_CONTENTS)
3730     {
3731       zfs_unmount (data);
3732       return grub_error (GRUB_ERR_BAD_FILE_TYPE, N_("not a regular file"));
3733     }

3735   /* get the file size and set the file position to 0 */

3737   /*
3738    * For DMU_OT_SA we will need to locate the SIZE attribute
3739    * attribute, which could be either in the bonus buffer
3740    * or the "spill" block.
3741    */
3742   if (data->dnode.dn.dn_bonustype == DMU_OT_SA)
3743     {
3744       void *sahdrp;
3745       int hdrsize;

3747       if (data->dnode.dn.dn_bonuslen != 0)
3748         {
3749           sahdrp = (sa_hdr_phys_t *) DN_BONUS (&data->dnode.dn);
3750         }
3751       else if (data->dnode.dn.dn_flags & DNODE_FLAG_SPILL_BLKPTR)
3752         {
3753           blkptr_t *bp = &data->dnode.dn.dn_spill;

3755           err = zio_read (bp, data->dnode.endian, &sahdrp, NULL, data);
3756           if (err)
3757             return err;
3758         }
3759       else
3760         {
3761           return grub_error (GRUB_ERR_BAD_FS, "filesystem is corrupt");
3762         }

3764       hdrsize = SA_HDR_SIZE (((sa_hdr_phys_t *) sahdrp));
3765       file->size = grub_zfs_to_cpu64 (grub_get_unaligned64 ((char *) sahdrp + hd
3766     }
3767   else if (data->dnode.dn.dn_bonustype == DMU_OT_ZNODE)
3768     {
3769       file->size = grub_zfs_to_cpu64 (((znode_phys_t *) DN_BONUS (&data->dnode.d
3770     }
3771   else
3772     return grub_error (GRUB_ERR_BAD_FS, "bad bonus type");

3774   file->data = data;
```



```
3775   file->offset = 0;

3777 #ifndef GRUB_UTIL
3778   grub_dl_ref (my_mod);
3779 #endif

3781   return GRUB_ERR_NONE;
3782 }

3784 static grub_ssize_t
3785 grub_zfs_read (grub_file_t file, char *buf, grub_size_t len)
3786 {
3787   struct grub_zfs_data *data = (struct grub_zfs_data *) file->data;
3788   grub_size_t blksz, movesize;
3789   grub_size_t length;
3790   grub_size_t read;
3791   grub_err_t err;

3793   /*
3794    * If offset is in memory, move it into the buffer provided and return.
3795    */
3796   if (file->offset >= data->file_start
3797       && file->offset + len <= data->file_end)
3798     {
3799       grub_memmove (buf, data->file_buf + file->offset - data->file_start,
3800                     len);
3801       return len;
3802     }

3804   blksz = grub_zfs_to_cpu16 (data->dnode.dn.dn_datablkszsec,
3805                              data->dnode.endian) << SPA_MINBLOCKSHIFT;

3807   /*
3808    * Entire Dnode is too big to fit into the space available.  We
3809    * will need to read it in chunks.  This could be optimized to
3810    * read in as large a chunk as there is space available, but for
3811    * now, this only reads in one data block at a time.
3812    */
3813   length = len;
3814   read = 0;
3815   while (length)
3816     {
3817       void *t;
3818       /*
3819        * Find requested blkid and the offset within that block.
3820        */
3821       grub_uint64_t blkid = grub_divmod64 (file->offset + read, blksz, 0);
3822       grub_free (data->file_buf);
3823       data->file_buf = 0;

3825       err = dmu_read (&(data->dnode), blkid, &t,
3826                       0, data);
3827       data->file_buf = t;
3828       if (err)
3829         {
3830           data->file_buf = NULL;
3831           data->file_start = data->file_end = 0;
3832           return -1;
3833         }

3835       data->file_start = blkid * blksz;
3836       data->file_end = data->file_start + blksz;

3838       movesize = data->file_end - file->offset - read;
3839       if (movesize > length)
3840         movesize = length;
```

```
3842       grub_memmove (buf, data->file_buf + file->offset + read
3843                     - data->file_start, movesize);
3844       buf += movesize;
3845       length -= movesize;
3846       read += movesize;
3847     }

3849   return len;
3850 }

3852 static grub_err_t
3853 grub_zfs_close (grub_file_t file)
3854 {
3855   zfs_unmount ((struct grub_zfs_data *) file->data);

3857 #ifndef GRUB_UTIL
3858   grub_dl_unref (my_mod);
3859 #endif

3861   return GRUB_ERR_NONE;
3862 }

3864 grub_err_t
3865 grub_zfs_getmdnobj (grub_device_t dev, const char *fsfilename,
3866                     grub_uint64_t *mdnobj)
3867 {
3868   struct grub_zfs_data *data;
3869   grub_err_t err;
3870   int isfs;

3872   data = zfs_mount (dev);
3873   if (! data)
3874     return grub_errno;

3876   err = dnode_get_fullpath (fsfilename, &(data->subvol),
3877                             &(data->dnode), &isfs, data);
3878   *mdnobj = data->subvol.obj;
3879   zfs_unmount (data);
3880   return err;
3881 }

3883 static void
3884 fill_fs_info (struct grub_dirhook_info *info,
3885               dnode_end_t mdn, struct grub_zfs_data *data)
3886 {
3887   grub_err_t err;
3888   dnode_end_t dn;
3889   grub_uint64_t objnum;
3890   grub_uint64_t headobj;
3891
3892   grub_memset (info, 0, sizeof (*info));
3893
3894   info->dir = 1;
3895
3896   if (mdn.dn.dn_type == DMU_OT_DSL_DIR)
3897     {
3898       headobj = grub_zfs_to_cpu64 (((dsl_dir_phys_t *) DN_BONUS (&mdn.dn))->dd_h

3900       err = dnode_get (&(data->mos), headobj, DMU_OT_DSL_DATASET, &mdn, data);
3901       if (err)
3902         {
3903           grub_dprintf ("zfs", "failed here\n");
3904           return;
3905         }
3906     }
```



```
3907   make_mdn (&mdn, data);
3908   err = dnode_get (&mdn, MASTER_NODE_OBJ, DMU_OT_MASTER_NODE,
3909                    &dn, data);
3910   if (err)
3911     {
3912       grub_dprintf ("zfs", "failed here\n");
3913       return;
3914     }
3915
3916   err = zap_lookup (&dn, ZFS_ROOT_OBJ, &objnum, data, 0);
3917   if (err)
3918     {
3919       grub_dprintf ("zfs", "failed here\n");
3920       return;
3921     }
3922
3923   err = dnode_get (&mdn, objnum, 0, &dn, data);
3924   if (err)
3925     {
3926       grub_dprintf ("zfs", "failed here\n");
3927       return;
3928     }
3929
3930   if (dn.dn.dn_bonustype == DMU_OT_SA)
3931     {
3932       void *sahdrp;
3933       int hdrsize;

3935       if (dn.dn.dn_bonuslen != 0)
3936         {
3937           sahdrp = (sa_hdr_phys_t *) DN_BONUS (&dn.dn);
3938         }
3939       else if (dn.dn.dn_flags & DNODE_FLAG_SPILL_BLKPTR)
3940         {
3941           blkptr_t *bp = &dn.dn.dn_spill;

3943           err = zio_read (bp, dn.endian, &sahdrp, NULL, data);
3944           if (err)
3945             return;
3946         }
3947       else
3948         {
3949           grub_error (GRUB_ERR_BAD_FS, "filesystem is corrupt");
3950           return;
3951         }

3953       hdrsize = SA_HDR_SIZE (((sa_hdr_phys_t *) sahdrp));
3954       info->mtimeset = 1;
3955       info->mtime = grub_zfs_to_cpu64 (grub_get_unaligned64 ((char *) sahdrp + h
3956     }

3958   if (dn.dn.dn_bonustype == DMU_OT_ZNODE)
3959     {
3960       info->mtimeset = 1;
3961       info->mtime = grub_zfs_to_cpu64 (((znode_phys_t *) DN_BONUS (&dn.dn))->zp_
3962     }
3963   return;
3964 }

3966 static grub_err_t
3967 grub_zfs_dir (grub_device_t device, const char *path,
3968               int (*hook) (const char *, const struct grub_dirhook_info *))
3969 {
3970   struct grub_zfs_data *data;
3971   grub_err_t err;
3972   int isfs;
```

```
3973  auto int NESTED_FUNC_ATTR iterate_zap (const char *name, grub_uint64_t val);
3974  auto int NESTED_FUNC_ATTR iterate_zap_fs (const char *name,
3975                                            grub_uint64_t val);
3976  auto int NESTED_FUNC_ATTR iterate_zap_snap (const char *name,
3977                                              grub_uint64_t val);

3979  int NESTED_FUNC_ATTR iterate_zap (const char *name, grub_uint64_t val)
3980  {
3981    struct grub_dirhook_info info;
3982    dnode_end_t dn;
3983    grub_memset (&info, 0, sizeof (info));

3985    dnode_get (&(data->subvol.mdn), val, 0, &dn, data);

3987    if (dn.dn.dn_bonustype == DMU_OT_SA)
3988      {
3989        void *sahdrp;
3990        int hdrsize;

3992        if (dn.dn.dn_bonuslen != 0)
3993          {
3994            sahdrp = (sa_hdr_phys_t *) DN_BONUS (&data->dnode.dn);
3995          }
3996        else if (dn.dn.dn_flags & DNODE_FLAG_SPILL_BLKPTR)
3997          {
3998            blkptr_t *bp = &dn.dn.dn_spill;

4000            err = zio_read (bp, dn.endian, &sahdrp, NULL, data);
4001            if (err)
4002              {
4003                grub_print_error ();
4004                return 0;
4005              }
4006          }
4007        else
4008          {
4009            grub_error (GRUB_ERR_BAD_FS, "filesystem is corrupt");
4010            grub_print_error ();
4011            return 0;
4012          }

4014        hdrsize = SA_HDR_SIZE (((sa_hdr_phys_t *) sahdrp));
4015        info.mtimeset = 1;
4016        info.mtime = grub_zfs_to_cpu64 (grub_get_unaligned64 ((char *) sahdrp +
4017        info.case_insensitive = data->subvol.case_insensitive;
4018      }
4019
4020    if (dn.dn.dn_bonustype == DMU_OT_ZNODE)
4021      {
4022        info.mtimeset = 1;
4023        info.mtime = grub_zfs_to_cpu64 (((znode_phys_t *) DN_BONUS (&dn.dn))->zp
4024                                        dn.endian);
4025      }
4026        info.dir = (dn.dn.dn_type == DMU_OT_DIRECTORY_CONTENTS);
4027        grub_dprintf ("zfs", "type=%d, name=%s\n",
4028                      (int)dn.dn.dn_type, (char *)name);
4029    return hook (name, &info);
4030  }

4032  int NESTED_FUNC_ATTR iterate_zap_fs (const char *name, grub_uint64_t val)
4033  {
4034    struct grub_dirhook_info info;
4035    dnode_end_t mdn;
4036    err = dnode_get (&(data->mos), val, 0, &mdn, data);
4037    if (err)
4038      return 0;
```



```
4039    if (mdn.dn.dn_type != DMU_OT_DSL_DIR)
4040      return 0;

4042    fill_fs_info (&info, mdn, data);
4043    return hook (name, &info);
4044  }
4045  int NESTED_FUNC_ATTR iterate_zap_snap (const char *name, grub_uint64_t val)
4046  {
4047    struct grub_dirhook_info info;
4048    char *name2;
4049    int ret;
4050    dnode_end_t mdn;

4052    err = dnode_get (&(data->mos), val, 0, &mdn, data);
4053    if (err)
4054      return 0;

4056    if (mdn.dn.dn_type != DMU_OT_DSL_DATASET)
4057      return 0;

4059    fill_fs_info (&info, mdn, data);

4061    name2 = grub_malloc (grub_strlen (name) + 2);
4062    name2[0] = '@';
4063    grub_memcpy (name2 + 1, name, grub_strlen (name) + 1);
4064    ret = hook (name2, &info);
4065    grub_free (name2);
4066    return ret;
4067  }

4069  data = zfs_mount (device);
4070  if (! data)
4071    return grub_errno;
4072  err = dnode_get_fullpath (path, &(data->subvol), &(data->dnode), &isfs, data);
4073  if (err)
4074    {
4075      zfs_unmount (data);
4076      return err;
4077    }
4078  if (isfs)
4079    {
4080      grub_uint64_t childobj, headobj;
4081      grub_uint64_t snapobj;
4082      dnode_end_t dn;
4083      struct grub_dirhook_info info;

4085      fill_fs_info (&info, data->dnode, data);
4086      hook ("@", &info);
4087
4088      childobj = grub_zfs_to_cpu64 (((dsl_dir_phys_t *) DN_BONUS (&data->dnode.d
4089      headobj = grub_zfs_to_cpu64 (((dsl_dir_phys_t *) DN_BONUS (&data->dnode.dn
4090      err = dnode_get (&(data->mos), childobj,
4091                       DMU_OT_DSL_DIR_CHILD_MAP, &dn, data);
4092      if (err)
4093        {
4094          zfs_unmount (data);
4095          return err;
4096        }

4098      zap_iterate_u64 (&dn, iterate_zap_fs, data);
4099
4100      err = dnode_get (&(data->mos), headobj, DMU_OT_DSL_DATASET, &dn, data);
4101      if (err)
4102        {
4103          zfs_unmount (data);
4104          return err;
```

```
4105         }

4107       snapobj = grub_zfs_to_cpu64 (((dsl_dataset_phys_t *) DN_BONUS (&dn.dn))->d

4109       err = dnode_get (&(data->mos), snapobj,
4110                        DMU_OT_DSL_DS_SNAP_MAP, &dn, data);
4111       if (err)
4112         {
4113           zfs_unmount (data);
4114           return err;
4115         }

4117       zap_iterate_u64 (&dn, iterate_zap_snap, data);
4118     }
4119   else
4120     {
4121       if (data->dnode.dn.dn_type != DMU_OT_DIRECTORY_CONTENTS)
4122         {
4123           zfs_unmount (data);
4124           return grub_error (GRUB_ERR_BAD_FILE_TYPE, N_("not a directory"));
4125         }
4126       zap_iterate_u64 (&(data->dnode), iterate_zap, data);
4127     }
4128   zfs_unmount (data);
4129   return grub_errno;
4130 }

4132 #ifdef GRUB_UTIL
4133 static grub_err_t
4134 grub_zfs_embed (grub_device_t device __attribute__ ((unused)),
4135                 unsigned int *nsectors,
4136                 unsigned int max_nsectors,
4137                 grub_embed_type_t embed_type,
4138                 grub_disk_addr_t **sectors)
4139 {
4140   unsigned i;

4142   if (embed_type != GRUB_EMBED_PCBIOS)
4143     return grub_error (GRUB_ERR_NOT_IMPLEMENTED_YET,
4144                        "ZFS currently supports only PC-BIOS embedding");

4146  if ((VDEV_BOOT_SIZE >> GRUB_DISK_SECTOR_BITS) < *nsectors)
4147     return grub_error (GRUB_ERR_OUT_OF_RANGE,
4148                        N_("your core.img is unusually large.  "
4149                           "It won't fit in the embedding area"));

4151   *nsectors = (VDEV_BOOT_SIZE >> GRUB_DISK_SECTOR_BITS);
4152   if (*nsectors > max_nsectors)
4153     *nsectors = max_nsectors;
4154   *sectors = grub_malloc (*nsectors * sizeof (**sectors));
4155   if (!*sectors)
4156     return grub_errno;
4157   for (i = 0; i < *nsectors; i++)
4158     (*sectors)[i] = i + (VDEV_BOOT_OFFSET >> GRUB_DISK_SECTOR_BITS);

4160   return GRUB_ERR_NONE;
4161 }
4162 #endif

4164 static struct grub_fs grub_zfs_fs = {
4165   .name = "zfs",
4166   .dir = grub_zfs_dir,
4167   .open = grub_zfs_open,
4168   .read = grub_zfs_read,
4169   .close = grub_zfs_close,
4170   .label = zfs_label,
```



```
4171   .uuid = zfs_uuid,
4172   .mtime = zfs_mtime,
4173 #ifdef GRUB_UTIL
4174   .embed = grub_zfs_embed,
4175   .reserved_first_sector = 1,
4176   .blocklist_install = 0,
4177 #endif
4178   .next = 0
4179 };

4181 GRUB_MOD_INIT (zfs)
4182 {
4183   COMPILE_TIME_ASSERT (sizeof (zap_leaf_chunk_t) == ZAP_LEAF_CHUNKSIZE);
4184   grub_fs_register (&grub_zfs_fs);
4185 #ifndef GRUB_UTIL
4186   my_mod = mod;
4187 #endif
4188 }

4190 GRUB_MOD_FINI (zfs)
4191 {
4192   grub_fs_unregister (&grub_zfs_fs);
4193 }
```

```
**********************************************************
   11496 Fri Aug 31 05:08:54 2012
new/grub/grub-core/fs/zfs/zfsinfo.c
grub patch
**********************************************************
______unchanged_portion_omitted_

 186 static grub_err_t
 187 get_bootpath (char *nvlist, char **bootpath, char **devid, grub_uint64_t inguid)
 187 get_bootpath (char *nvlist, char **bootpath, char **devid)
 188 {
 189   char *type = 0;
 190   grub_uint64_t diskguid = 0;
 191 #endif /* ! codereview */

 193   type = grub_zfs_nvlist_lookup_string (nvlist, ZPOOL_CONFIG_TYPE);

 195   if (!type)
 196     return grub_errno;

 198   if (grub_strcmp (type, VDEV_TYPE_DISK) == 0)
 199     {
 200       grub_zfs_nvlist_lookup_uint64 (nvlist, "guid", &diskguid);
 201 #endif /* ! codereview */
 202       *bootpath = grub_zfs_nvlist_lookup_string (nvlist,
 203                                                  ZPOOL_CONFIG_PHYS_PATH);
 204       *devid = grub_zfs_nvlist_lookup_string (nvlist, ZPOOL_CONFIG_DEVID);
 205       if (!*bootpath || !*devid || (diskguid != inguid))
 190       if (!*bootpath || !*devid)
 206         {
 207           grub_free (*bootpath);
 208           grub_free (*devid);
 209           *bootpath = 0;
 210           *devid = 0;
 211         }
 212       return GRUB_ERR_NONE;
 213     }

 215   if (grub_strcmp (type, VDEV_TYPE_MIRROR) == 0)
 216     {
 217       int nelm, i;

 219       nelm = grub_zfs_nvlist_lookup_nvlist_array_get_nelm
 220         (nvlist, ZPOOL_CONFIG_CHILDREN);

 222       for (i = 0; i < nelm; i++)
 223         {
 224           char *child;

 226           child = grub_zfs_nvlist_lookup_nvlist_array (nvlist,
 227                                                        ZPOOL_CONFIG_CHILDREN,
 228                                                        i);

 230           get_bootpath (child, bootpath, devid, inguid);
 215           get_bootpath (child, bootpath, devid);

 232           grub_free (child);

 234           if (*bootpath && *devid)
 235             return GRUB_ERR_NONE;
 236         }
 237     }

 239   return GRUB_ERR_NONE;
 240 }
______unchanged_portion_omitted_
```



```
 334 static grub_err_t
 335 grub_cmd_zfs_bootfs (grub_command_t cmd __attribute__ ((unused)), int argc,
 336                      char **args)
 337 {
 338   grub_device_t dev;
 339   char *devname;
 340   grub_err_t err;
 341   char *nvlist = 0;
 342   char *nv = 0;
 343   char *bootpath = 0, *devid = 0;
 344   char *fsname;
 345   char *bootfs;
 346   char *poolname;
 347   char def_path[512];
 348   struct grub_zfs_data * data;
 349   grub_uint64_t mdnobj, guid;
 350   int def = 0;
 332   grub_uint64_t mdnobj;

 352   if (argc <  1)
 353     return grub_error (GRUB_ERR_BAD_ARGUMENT, N_("one argument expected"));

 355   devname = grub_file_get_device_name (args[0]);

 357 #endif /* ! codereview */
 358   if (grub_errno)
 359     return grub_errno;

 361   dev = grub_device_open (devname);
 362   grub_free (devname);
 363   if (!dev)
 364     return grub_errno;

 366   err = grub_zfs_fetch_nvlist (dev, &nvlist);

 368   fsname = grub_strchr (args[0], ')');
 369   if (fsname)
 370     fsname++;
 371   else
 372     fsname = args[0];

 374   if (grub_strcmp(fsname, "default") == 0)
 375     ++def;

 377   if (! def) {
 338   if (!err)
 378     err = grub_zfs_getmdnobj (dev, fsname, &mdnobj);
 379   } else {
 380     err = get_default_bootfs_obj(dev, def_path, &mdnobj);
 381   }
 382 #endif /* ! codereview */

 384   grub_device_close (dev);

 386   if (err)
 387     return err;

 389   poolname = grub_zfs_nvlist_lookup_string (nvlist, ZPOOL_CONFIG_POOL_NAME);
 390   if (!poolname)
 391     {
 392       if (!grub_errno)
 393         grub_error (GRUB_ERR_BAD_FS, "No poolname found");
 394       return grub_errno;
 395     }
```

```
  nv = grub_zfs_nvlist_lookup_nvlist (nvlist, ZPOOL_CONFIG_VDEV_TREE);
  grub_zfs_nvlist_lookup_uint64 (nvlist, "guid", &guid);
#endif /* ! codereview */

  if (nv && guid)
    get_bootpath (nv, &bootpath, &devid, guid);
  if (nv)
    get_bootpath (nv, &bootpath, &devid);

  grub_free (nv);
  grub_free (nvlist);

  bootfs = grub_xasprintf ("zfs-bootfs=%s/%llu%s%s%s%s%s%s",
                           poolname, (unsigned long long) mdnobj,
                           bootpath ? ",bootpath=\"" : "",
                           bootpath ? : "",
                           bootpath ? "\"" : "",
                           devid ? ",diskdevid=\"" : "",
                           devid ? : "",
                           devid ? "\"" : "");
  if (!bootfs)
    return grub_errno;
  if (argc >= 2)
    grub_env_set (args[1], bootfs);
  else
    grub_printf ("%s\n", bootfs);

  if ((def) && (argc >= 3))
    grub_env_set(args[2], def_path);
  else if (def)
    grub_printf("%s\n", def_path);

#endif /* ! codereview */
  grub_free (bootfs);
  grub_free (poolname);
  grub_free (bootpath);
  grub_free (devid);

  return GRUB_ERR_NONE;
}


static grub_command_t cmd_info, cmd_bootfs;

GRUB_MOD_INIT (zfsinfo)
{
  cmd_info = grub_register_command ("zfsinfo", grub_cmd_zfsinfo,
                                    N_("DEVICE"),
                                    N_("Print ZFS info about DEVICE."));
  cmd_bootfs = grub_register_command ("zfs-bootfs", grub_cmd_zfs_bootfs,
                                      N_("FILESYSTEM [VARIABLE]"),
                                      N_("Print ZFS-BOOTFSOBJ or store it into V
}

GRUB_MOD_FINI (zfsinfo)
{
  grub_unregister_command (cmd_info);
  grub_unregister_command (cmd_bootfs);
}
```

****
4574 Fri Aug 31 05:08:55 2012
new/grub/grub-core/partmap/sun.c
grub patch
****
_____unchanged_portion_omitted_

```
 87 static grub_err_t
 88 sun_partition_map_iterate (grub_disk_t disk,
 89                           int (*hook) (grub_disk_t disk,
 90                                        const grub_partition_t partition))
 91 {
 92   struct grub_partition p;
 93   union
 94   {
 95     struct grub_sun_block sunb;
 95     struct grub_sun_block sun;
 96     grub_uint16_t raw[0];
 97   } block;
 98   int partnum;
 99   grub_err_t err;

101   p.partmap = &grub_sun_partition_map;
102   err = grub_disk_read (disk, 0, 0, sizeof (struct grub_sun_block),
103                         &block);
104   if (err)
105     return err;

107   if (GRUB_PARTMAP_SUN_MAGIC != grub_be_to_cpu16 (block.sunb.magic))
107   if (GRUB_PARTMAP_SUN_MAGIC != grub_be_to_cpu16 (block.sun.magic))
108     return grub_error (GRUB_ERR_BAD_PART_TABLE, "not a sun partition table");

110   if (! grub_sun_is_valid (block.raw))
111       return grub_error (GRUB_ERR_BAD_PART_TABLE, "invalid checksum");
112
113   /* Maybe another error value would be better, because partition
114      table _is_ recognized but invalid.  */
115   for (partnum = 0; partnum < GRUB_PARTMAP_SUN_MAX_PARTS; partnum++)
116     {
117       struct grub_sun_partition_descriptor *desc;

119       if (block.sunb.infos[partnum].id == 0
120           || block.sunb.infos[partnum].id == GRUB_PARTMAP_SUN_WHOLE_DISK_ID)
119       if (block.sun.infos[partnum].id == 0
120           || block.sun.infos[partnum].id == GRUB_PARTMAP_SUN_WHOLE_DISK_ID)
121         continue;

123       desc = &block.sunb.partitions[partnum];
123       desc = &block.sun.partitions[partnum];
124       p.start = ((grub_uint64_t) grub_be_to_cpu32 (desc->start_cylinder)
125                  * grub_be_to_cpu16 (block.sunb.ntrks)
126                  * grub_be_to_cpu16 (block.sunb.nsect));
125                  * grub_be_to_cpu16 (block.sun.ntrks)
126                  * grub_be_to_cpu16 (block.sun.nsect));
127       p.len = grub_be_to_cpu32 (desc->num_sectors);
128       p.number = p.index = partnum;
129       if (p.len)
130         {
131           if (hook (disk, &p))
132             partnum = GRUB_PARTMAP_SUN_MAX_PARTS;
133         }
134     }

136   return grub_errno;
137 }
```

_____unchanged_portion_omitted_

****
3952 Fri Aug 31 05:08:55 2012
new/grub/grub-core/partmap/sunpc.c
grub patch
****
_____***unchanged_portion_omitted_***

```
 69 static grub_err_t
 70 sun_pc_partition_map_iterate (grub_disk_t disk,
 71                               int (*hook) (grub_disk_t disk,
 72                                            const grub_partition_t partition))
 73 {
 74   grub_partition_t p;
 75   union
 76   {
 77     struct grub_sun_pc_block sunb;
 77     struct grub_sun_pc_block sun;
 78     grub_uint16_t raw[0];
 79   } block;
 80   int partnum;
 81   grub_err_t err;

 83   p = (grub_partition_t) grub_zalloc (sizeof (struct grub_partition));
 84   if (! p)
 85     return grub_errno;

 87   p->partmap = &grub_sun_pc_partition_map;
 88   err = grub_disk_read (disk, 1, 0, sizeof (struct grub_sun_pc_block), &block);
 89   if (err)
 90     {
 91       grub_free (p);
 92       return err;
 93     }
 94
 95   if (GRUB_PARTMAP_SUN_PC_MAGIC != grub_le_to_cpu16 (block.sunb.magic))
 95   if (GRUB_PARTMAP_SUN_PC_MAGIC != grub_le_to_cpu16 (block.sun.magic))
 96     {
 97       grub_free (p);
 98       return grub_error (GRUB_ERR_BAD_PART_TABLE,
 99                          "not a sun_pc partition table");
100     }

102   if (! grub_sun_is_valid (block.raw))
103     {
104       grub_free (p);
105       return grub_error (GRUB_ERR_BAD_PART_TABLE, "invalid checksum");
106     }

108   /* Maybe another error value would be better, because partition
109      table _is_ recognized but invalid.  */
110   for (partnum = 0; partnum < GRUB_PARTMAP_SUN_PC_MAX_PARTS; partnum++)
111     {
112       struct grub_sun_pc_partition_descriptor *desc;

114       if (block.sunb.partitions[partnum].id == 0
115           || block.sunb.partitions[partnum].id
114       if (block.sun.partitions[partnum].id == 0
115           || block.sun.partitions[partnum].id
116           == GRUB_PARTMAP_SUN_PC_WHOLE_DISK_ID)
117         continue;

119       desc = &block.sunb.partitions[partnum];
119       desc = &block.sun.partitions[partnum];
120       p->start = grub_le_to_cpu32 (desc->start_sector);
121       p->len = grub_le_to_cpu32 (desc->num_sectors);
122       p->number = partnum;
```



```
123       if (p->len)
124         {
125           if (hook (disk, p))
126             partnum = GRUB_PARTMAP_SUN_PC_MAX_PARTS;
127         }
128     }

130   grub_free (p);

132   return grub_errno;
133 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
    1170 Fri Aug 31 05:08:56 2012
new/grub/include/grub/search.h
fix mirror
**********************************************************
```

```
/*
 *  GRUB  --  GRand Unified Bootloader
 *  Copyright (C) 2009  Free Software Foundation, Inc.
 *
 *  GRUB is free software: you can redistribute it and/or modify
 *  it under the terms of the GNU General Public License as published by
 *  the Free Software Foundation, either version 3 of the License, or
 *  (at your option) any later version.
 *
 *  GRUB is distributed in the hope that it will be useful,
 *  but WITHOUT ANY WARRANTY; without even the implied warranty of
 *  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 *  GNU General Public License for more details.
 *
 *  You should have received a copy of the GNU General Public License
 *  along with GRUB.  If not, see <http://www.gnu.org/licenses/>.
 */

#ifndef GRUB_SEARCH_HEADER
#define GRUB_SEARCH_HEADER 1

void grub_search_fs_file (const char *key, const char *var, int no_floppy,
                          char **hints, unsigned nhints, int mirror);
                          char **hints, unsigned nhints);
void grub_search_fs_uuid (const char *key, const char *var, int no_floppy,
                          char **hints, unsigned nhints, int mirror);
                          char **hints, unsigned nhints);
void grub_search_label (const char *key, const char *var, int no_floppy,
                        char **hints, unsigned nhints, int mirror);
                        char **hints, unsigned nhints);

#endif
```

```
**********************************************************
    4683 Fri Aug 31 05:08:56 2012
new/grub/include/grub/zfs/dmu.h
feature flags + bug fix
**********************************************************
/*
 *  GRUB  --  GRand Unified Bootloader
 *  Copyright (C) 1999,2000,2001,2002,2003,2004  Free Software Foundation, Inc.
 *
 *  GRUB is free software; you can redistribute it and/or modify
 *  it under the terms of the GNU General Public License as published by
 *  the Free Software Foundation; either version 3 of the License, or
 *  (at your option) any later version.
 *
 *  GRUB is distributed in the hope that it will be useful,
 *  but WITHOUT ANY WARRANTY; without even the implied warranty of
 *  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 *  GNU General Public License for more details.
 *
 *  You should have received a copy of the GNU General Public License
 *  along with GRUB.  If not, see <http://www.gnu.org/licenses/>.
 */
/*
 * Copyright 2010 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 */
/*
 *  Copyright 2012, Daniil Lunev
 */
#endif /* ! codereview */

#ifndef _SYS_DMU_H
#define _SYS_DMU_H

/*
 * This file describes the interface that the DMU provides for its
 * consumers.
 *
 * The DMU also interacts with the SPA.  That interface is described in
 * dmu_spa.h.
 */

#define  B_FALSE  0
#define  B_TRUE  1

#define  DMU_OT_NEWTYPE 0x80
#define  DMU_OT_METADATA 0x40
#define  DMU_OT_BYTESWAP_MASK 0x3f

#define  DMU_OT(byteswap, metadata) \
  (DMU_OT_NEWTYPE | \
  ((metadata) ? DMU_OT_METADATA : 0) | \
  ((byteswap) & DMU_OT_BYTESWAP_MASK))

#define  DMU_OT_IS_VALID(ot) (((ot) & DMU_OT_NEWTYPE) ? \
  ((ot) & DMU_OT_BYTESWAP_MASK) < DMU_BSWAP_NUMFUNCS : \
  (ot) < DMU_OT_NUMTYPES)

#define  DMU_OT_IS_METADATA(ot) (((ot) & DMU_OT_NEWTYPE) ? \
  ((ot) & DMU_OT_METADATA) : \
  dmu_ot[(ot)].ot_metadata)

typedef enum dmu_object_byteswap {
  DMU_BSWAP_UINT8,
  DMU_BSWAP_UINT16,
  DMU_BSWAP_UINT32,
```



```
  DMU_BSWAP_UINT64,
  DMU_BSWAP_ZAP,
  DMU_BSWAP_DNODE,
  DMU_BSWAP_OBJSET,
  DMU_BSWAP_ZNODE,
  DMU_BSWAP_OLDACL,
  DMU_BSWAP_ACL,
  DMU_BSWAP_NUMFUNCS
} dmu_object_byteswap_t;

#endif /* ! codereview */
typedef enum dmu_object_type {
        DMU_OT_NONE,
        /* general: */
        DMU_OT_OBJECT_DIRECTORY,        /* ZAP */
        DMU_OT_OBJECT_ARRAY,            /* UINT64 */
        DMU_OT_PACKED_NVLIST,           /* UINT8 (XDR by nvlist_pack/unpack) */
        DMU_OT_PACKED_NVLIST_SIZE,      /* UINT64 */
        DMU_OT_BPLIST,                  /* UINT64 */
        DMU_OT_BPLIST_HDR,              /* UINT64 */
        /* spa: */
        DMU_OT_SPACE_MAP_HEADER,        /* UINT64 */
        DMU_OT_SPACE_MAP,               /* UINT64 */
        /* zil: */
        DMU_OT_INTENT_LOG,              /* UINT64 */
        /* dmu: */
        DMU_OT_DNODE,                   /* DNODE */
        DMU_OT_OBJSET,                  /* OBJSET */
        /* dsl: */
        DMU_OT_DSL_DIR,                 /* UINT64 */
        DMU_OT_DSL_DIR_CHILD_MAP,       /* ZAP */
        DMU_OT_DSL_DS_SNAP_MAP,         /* ZAP */
        DMU_OT_DSL_PROPS,               /* ZAP */
        DMU_OT_DSL_DATASET,             /* UINT64 */
        /* zpl: */
        DMU_OT_ZNODE,                   /* ZNODE */
        DMU_OT_OLDACL,                  /* OLD ACL */
        DMU_OT_PLAIN_FILE_CONTENTS,     /* UINT8 */
        DMU_OT_DIRECTORY_CONTENTS,      /* ZAP */
        DMU_OT_MASTER_NODE,             /* ZAP */
        DMU_OT_UNLINKED_SET,            /* ZAP */
        /* zvol: */
        DMU_OT_ZVOL,                    /* UINT8 */
        DMU_OT_ZVOL_PROP,               /* ZAP */
        /* other; for testing only! */
        DMU_OT_PLAIN_OTHER,             /* UINT8 */
        DMU_OT_UINT64_OTHER,            /* UINT64 */
        DMU_OT_ZAP_OTHER,               /* ZAP */
        /* new object types: */
        DMU_OT_ERROR_LOG,               /* ZAP */
        DMU_OT_SPA_HISTORY,             /* UINT8 */
        DMU_OT_SPA_HISTORY_OFFSETS,     /* spa_his_phys_t */
        DMU_OT_POOL_PROPS,              /* ZAP */
        DMU_OT_DSL_PERMS,               /* ZAP */
        DMU_OT_ACL,                     /* ACL */
        DMU_OT_SYSACL,                  /* SYSACL */
        DMU_OT_FUID,                    /* FUID table (Packed NVLIST UINT8) */
        DMU_OT_FUID_SIZE,               /* FUID table size UINT64 */
        DMU_OT_NEXT_CLONES,             /* ZAP */
        DMU_OT_SCRUB_QUEUE,             /* ZAP */
        DMU_OT_USERGROUP_USED,          /* ZAP */
        DMU_OT_USERGROUP_QUOTA,         /* ZAP */
        DMU_OT_USERREFS,                /* ZAP */
        DMU_OT_DDT_ZAP,                 /* ZAP */
        DMU_OT_DDT_STATS,               /* ZAP */
        DMU_OT_SA,                      /* System attr */
```

```
128         DMU_OT_SA_MASTER_NODE,          /* ZAP */
129         DMU_OT_SA_ATTR_REGISTRATION,    /* ZAP */
130         DMU_OT_SA_ATTR_LAYOUTS,         /* ZAP */
131         DMU_OT_DSL_KEYCHAIN = 54,
132         DMU_OT_NUMTYPES,
133
134         DMU_OTN_ZAP_METADATA = DMU_OT(DMU_BSWAP_ZAP, B_TRUE),
 22         DMU_OT_NUMTYPES
135 } dmu_object_type_t;
```

*unchanged_portion_omitted_*

```
147 /*
148  * The names of zap entries in the DIRECTORY_OBJECT of the MOS.
149  */
150 #define DMU_POOL_DIRECTORY_OBJECT       1
151 #define DMU_POOL_CONFIG                 "config"
152 #define DMU_POOL_ROOT_DATASET           "root_dataset"
153 #define DMU_POOL_SYNC_BPLIST            "sync_bplist"
154 #define DMU_POOL_ERRLOG_SCRUB           "errlog_scrub"
155 #define DMU_POOL_ERRLOG_LAST            "errlog_last"
156 #define DMU_POOL_SPARES                 "spares"
157 #define DMU_POOL_DEFLATE                "deflate"
158 #define DMU_POOL_HISTORY                "history"
159 #define DMU_POOL_PROPS                  "pool_props"
160 #define DMU_POOL_L2CACHE                "l2cache"
161 #define DMU_POOL_FEATURES_FOR_READ      "features_for_read"
162 #endif /* ! codereview */

164 #endif  /* _SYS_DMU_H */
```

```
**********************************************************
    5552 Fri Aug 31 05:08:57 2012
new/grub/include/grub/zfs/zfs.h
grub patch
**********************************************************
```

```
/*
 *  GRUB  --  GRand Unified Bootloader
 *  Copyright (C) 1999,2000,2001,2002,2003,2004,2009  Free Software Foundation,
 *
 *  GRUB is free software; you can redistribute it and/or modify
 *  it under the terms of the GNU General Public License as published by
 *  the Free Software Foundation; either version 3 of the License, or
 *  (at your option) any later version.
 *
 *  GRUB is distributed in the hope that it will be useful,
 *  but WITHOUT ANY WARRANTY; without even the implied warranty of
 *  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 *  GNU General Public License for more details.
 *
 *  You should have received a copy of the GNU General Public License
 *  along with GRUB.  If not, see <http://www.gnu.org/licenses/>.
 */
/*
 * Copyright (c) 2007, 2010, Oracle and/or its affiliates. All rights reserved.
 */
/*
 *  Copyright 2012, Daniil Lunev
 */
#endif /* ! codereview */

#ifndef GRUB_ZFS_HEADER
#define GRUB_ZFS_HEADER 1

#include <grub/err.h>
#include <grub/disk.h>
#include <grub/crypto.h>

typedef enum grub_zfs_endian
  {
    GRUB_ZFS_UNKNOWN_ENDIAN = -2,
    GRUB_ZFS_LITTLE_ENDIAN = -1,
    GRUB_ZFS_BIG_ENDIAN = 0
  } grub_zfs_endian_t;

/*
 * On-disk version number.
 */
#define SPA_VERSION                     33ULL
#define SPA_FEATURE_VERSION             5000ULL
#endif /* ! codereview */

/*
 * The following are configuration names used in the nvlist describing a pool's
 * configuration.
 */
#define ZPOOL_CONFIG_VERSION            "version"
#define ZPOOL_CONFIG_POOL_NAME          "name"
#define ZPOOL_CONFIG_POOL_STATE         "state"
#define ZPOOL_CONFIG_POOL_TXG           "txg"
#define ZPOOL_CONFIG_POOL_GUID          "pool_guid"
#define ZPOOL_CONFIG_CREATE_TXG         "create_txg"
#define ZPOOL_CONFIG_TOP_GUID           "top_guid"
#define ZPOOL_CONFIG_VDEV_TREE          "vdev_tree"
#define ZPOOL_CONFIG_TYPE               "type"
#define ZPOOL_CONFIG_CHILDREN           "children"
#define ZPOOL_CONFIG_ID                 "id"
```



```
#define ZPOOL_CONFIG_GUID               "guid"
#define ZPOOL_CONFIG_PATH               "path"
#define ZPOOL_CONFIG_DEVID              "devid"
#define ZPOOL_CONFIG_METASLAB_ARRAY     "metaslab_array"
#define ZPOOL_CONFIG_METASLAB_SHIFT     "metaslab_shift"
#define ZPOOL_CONFIG_ASHIFT             "ashift"
#define ZPOOL_CONFIG_ASIZE              "asize"
#define ZPOOL_CONFIG_DTL                "DTL"
#define ZPOOL_CONFIG_STATS              "stats"
#define ZPOOL_CONFIG_WHOLE_DISK         "whole_disk"
#define ZPOOL_CONFIG_ERRCOUNT           "error_count"
#define ZPOOL_CONFIG_NOT_PRESENT        "not_present"
#define ZPOOL_CONFIG_SPARES             "spares"
#define ZPOOL_CONFIG_IS_SPARE           "is_spare"
#define ZPOOL_CONFIG_NPARITY            "nparity"
#define ZPOOL_CONFIG_PHYS_PATH          "phys_path"
#define ZPOOL_CONFIG_L2CACHE            "l2cache"
#define ZPOOL_CONFIG_HOLE_ARRAY         "hole_array"
#define ZPOOL_CONFIG_VDEV_CHILDREN      "vdev_children"
#define ZPOOL_CONFIG_IS_HOLE            "is_hole"
#define ZPOOL_CONFIG_DDT_HISTOGRAM      "ddt_histogram"
#define ZPOOL_CONFIG_DDT_OBJ_STATS      "ddt_object_stats"
#define ZPOOL_CONFIG_DDT_STATS          "ddt_stats"
/*
 * The persistent vdev state is stored as separate values rather than a single
 * 'vdev_state' entry.  This is because a device can be in multiple states, such
 * as offline and degraded.
 */
#define ZPOOL_CONFIG_OFFLINE            "offline"
#define ZPOOL_CONFIG_FAULTED            "faulted"
#define ZPOOL_CONFIG_DEGRADED           "degraded"
#define ZPOOL_CONFIG_REMOVED            "removed"

#define VDEV_TYPE_ROOT                  "root"
#define VDEV_TYPE_MIRROR                "mirror"
#define VDEV_TYPE_REPLACING             "replacing"
#define VDEV_TYPE_RAIDZ                 "raidz"
#define VDEV_TYPE_DISK                  "disk"
#define VDEV_TYPE_FILE                  "file"
#define VDEV_TYPE_MISSING               "missing"
#define VDEV_TYPE_HOLE                  "hole"
#define VDEV_TYPE_SPARE                 "spare"
#define VDEV_TYPE_L2CACHE               "l2cache"

/*
 * pool state.  The following states are written to disk as part of the normal
 * SPA lifecycle: ACTIVE, EXPORTED, DESTROYED, SPARE, L2CACHE.  The remaining
 * states are software abstractions used at various levels to communicate pool
 * state.
 */
typedef enum pool_state {
        POOL_STATE_ACTIVE = 0,          /* In active use                */
        POOL_STATE_EXPORTED,            /* Explicitly exported          */
        POOL_STATE_DESTROYED,           /* Explicitly destroyed         */
        POOL_STATE_SPARE,               /* Reserved for hot spare use   */
        POOL_STATE_L2CACHE,             /* Level 2 ARC device           */
        POOL_STATE_UNINITIALIZED,       /* Internal spa_t state         */
        POOL_STATE_UNAVAIL,             /* Internal libzfs state        */
        POOL_STATE_POTENTIALLY_ACTIVE   /* Internal libzfs state        */
} pool_state_t;

struct grub_zfs_data;

grub_err_t grub_zfs_fetch_nvlist (grub_device_t dev, char **nvlist);
grub_err_t grub_zfs_getmdnobj (grub_device_t dev, const char *fsfilename,
                               grub_uint64_t *mdnobj);
```

```
129 char *grub_zfs_nvlist_lookup_string (const char *nvlist, const char *name);
130 char *grub_zfs_nvlist_lookup_nvlist (const char *nvlist, const char *name);
131 int grub_zfs_nvlist_lookup_uint64 (const char *nvlist, const char *name,
132                                    grub_uint64_t *out);
133 char *grub_zfs_nvlist_lookup_nvlist_array (const char *nvlist,
134                                            const char *name,
135                                            grub_size_t index);
136 int grub_zfs_nvlist_lookup_nvlist_array_get_nelm (const char *nvlist,
137                                                   const char *name);
138 grub_err_t
139 grub_zfs_add_key (grub_uint8_t *key_in,
140                   grub_size_t keylen,
141                   int passphrase);

143 grub_err_t
144 get_default_bootfs_obj(grub_device_t dev, char * path, grub_uint64_t * mdnobj);
145 #endif /* ! codereview */
146 extern grub_err_t (*grub_zfs_decrypt) (grub_crypto_cipher_handle_t cipher,
147                                        grub_uint64_t algo,
148                                        void *nonce,
149                                        char *buf, grub_size_t size,
150                                        const grub_uint32_t *expected_mac,
151                                        grub_zfs_endian_t endian);

153 struct grub_zfs_key;

155 extern grub_crypto_cipher_handle_t (*grub_zfs_load_key) (const struct grub_zfs_k
156                                                          grub_size_t keysize,
157                                                          grub_uint64_t salt,
158                                                          grub_uint64_t algo);



162 #endif  /* ! GRUB_ZFS_HEADER */
```

```
**********************************************************
    5580 Fri Aug 31 05:08:58 2012
new/grub/util/grub-solarislst2cfg.c
grub patch
**********************************************************
```

```
/*
 *  GRUB  --  GRand Unified Bootloader
 *  Copyright (C) 2012 Daniil Lunev
 *
 *  GRUB is free software: you can redistribute it and/or modify
 *  it under the terms of the GNU General Public License as published by
 *  the Free Software Foundation, either version 3 of the License, or
 *  (at your option) any later version.
 *
 *  GRUB is distributed in the hope that it will be useful,
 *  but WITHOUT ANY WARRANTY; without even the implied warranty of
 *  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 *  GNU General Public License for more details.
 *
 *  You should have received a copy of the GNU General Public License
 *  along with GRUB.  If not, see <http://www.gnu.org/licenses/>.
 */

#include <grub/types.h>
#include <grub/emu/misc.h>
#include <grub/emu/getroot.h>
#include <grub/fs.h>
#include <grub/device.h>

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define BUFFER_SIZE 512

struct commands {
  const char * lst;
  const char * cfg;
};

void
get_uuid_by_path(char * path, char ** uuid)
{
  char ** devices = NULL;
  char * drive = NULL;
  grub_device_t dev = NULL;
  grub_fs_t fs = NULL;
  char * grub_path = NULL;

  grub_path = canonicalize_file_name(path);
  if (! grub_path)
    grub_util_error("Can't canonicalize path");

  devices = grub_guess_root_devices(grub_path);
  if (! devices)
    grub_util_error("Can't find device");

  drive = grub_util_get_grub_dev(devices[0]);
  if (! drive)
    grub_util_error("Can't get drive");

  dev = grub_device_open(drive);
  if (! dev)
    grub_util_error("Can't open device");

  fs = grub_fs_probe(dev);
```



```
  if (! fs)
    grub_util_error("Probing fs error");

  if (! fs->uuid)
    grub_util_error("This fs doesn't support uuid");

  if (fs->uuid(dev, uuid) != GRUB_ERR_NONE)
    grub_util_error("%s", grub_errmsg);

  free(grub_path);
  if (dev)
    grub_device_close(dev);

  return;
}

struct commands com[] = {
  {"title", "entry_name"},
  {"findroot", "pool"},
  {"bootfs", "data_set"},
  {"kernel$", "kernel_path"},
  {"module$", "module"},
  {"default", "default_entry"},
  {"timeout", "timeout"},
  {"serial", "serial"},
  {"terminal", "terminal"},
  {0, 0},
};

int
get_command_id(char * command)
{
  int i = 0;
  for (; com[i].lst; ++i)
    if (!strcmp(com[i].lst, command))
      return i;
  return -1;
}

char *
retrive_value(char * val)
{
  char * str;
  int quote_flag = 0;;

  while ((*val == ' ') || (*val == '\t'))
    ++val;

  if (*val == '"') {
    ++val;
    ++quote_flag;
  }

  str = val;

  for (;;) {
    if (quote_flag) {
      if (*val == '"')
        break;
      if (*val == '\0')
        return NULL;
    } else {
      if ((*val == ' ') || (*val == '\t') || (*val == '\0'))
        break;
    }
    ++val;
```

```
128   }
129   *val = '\0';

131   return str;
132 }

134 int
135 parse_menulst(FILE * in_file, FILE * out_file)
136 {
137   char line[BUFFER_SIZE];
138   char * command;
139   char * value;
140   char * label = NULL;
141   char bootfs[BUFFER_SIZE];
142   char pool[BUFFER_SIZE];
143   char * uuid;
144   char * c;
145   int com_id;

147   for (;;) {
148     command = fgets(line, BUFFER_SIZE, in_file);
149     if (! command)
150       return 0;

152     c = command + strlen(command) - 1;
153     if ((*c != '\n') && (fgets(line, BUFFER_SIZE, in_file)))
154       return 1;

156     if ((*command == '#') || (*command == ' ')
157        || (*command == '\t') || (*command == '\n'))
158       continue;

160     if (*c == '\n')
161       *c = '\0';
162     value = strchr(line, ' ');
163     if (! value)
164       continue;

166     *value++ = '\0';
167
168     com_id = get_command_id(command);
169     if (com_id < 0)
170       continue;

172     switch (com_id) {
173     case 0:
174       fprintf(out_file,"\n%s=%s\n", com[com_id].cfg, value);
175       break;
176     case 1:
177       label = strdup(strchr(value, '_') + 1);
178       c = strchr(label, ',');
179       if (c)
180         *c = 0;
181       continue;
182     case 2:
183       c = strchr(value, '/');
184       if (c != NULL) {
185         strcpy(bootfs, c);
186         *c = '\0';
187         strcpy(pool + 1, value);
188         *pool = '/';
189         uuid = NULL;
190         get_uuid_by_path(pool, &uuid);
191         if (! uuid)
192           return 1;
193
```



```
194         fprintf(out_file,"pool_uuid=%s\n", uuid);
195
196       }
197       *c = '/';
198       fprintf(out_file,"data_set=%s\n", value);
199       if (label) {
200         free(label);
201         label = NULL;
202       }
203       break;
204     case 3:
205     case 4:
206       if (label) {
207         fprintf(out_file,"pool_label=%s\n", label);
208         free(label);
209         label = NULL;
210       }
211       if (com_id == 3)
212         c = strchr (value, ' ');
213         if (c)
214           *c = '\0';
215       if (! value)
216         return 1;
217
218       fprintf(out_file,"%s=%s\n", com[com_id].cfg, value);

220       if ((com_id == 3) && (c)) {
221         char * tmp;
222         value = c + 1;
223         tmp = strstr(value, "ZFS-BOOTFS");
224         if (tmp)
225           *(tmp + 3) = '_';
226         fprintf(out_file,"kernel_options=%s\n", value);
227       }

229       break;
230     default:
231       fprintf(out_file,"%s=%s\n", com[com_id].cfg, value);
232       break;
233     }
234   }
235   return 0;
236 }

238 int
239 main(int argc, char ** argv)
240 {
241   FILE * lst_file;
242   FILE * cfg_file;
243   int err = 0;

245   if (argc != 3) {
246     printf("grub-solarislst2cfg lst_file cfg_file\n");
247     return 1;
248   }

250   lst_file = fopen(argv[1], "r");
251   if (! lst_file)
252     return 1;

254   cfg_file = fopen(argv[2], "w");
255   if (! cfg_file) {
256     fclose(lst_file);
257     return 1;
258   }
```

```
260   grub_util_init_nls();
261   grub_util_biosdisk_init(DEFAULT_DEVICE_MAP);
262   grub_init_all();
263
264   err = parse_menulst(lst_file, cfg_file);

266   grub_fini_all();

268   fclose(lst_file);
269   fclose(cfg_file);

271   if (err)
272     remove(argv[2]);
273   return err;
274 }
275 #endif /* ! codereview */
```

****
1774 Fri Aug 31 05:08:58 2012
new/grub/util/grub.d/10_illumos.in
fixes
****
_____unchanged_portion_omitted_

```
 53 submenu 'Illumos-entries' {
 54         illumos_entries /@/boot/illumos.cfg
 55 }
 56 #endif /* ! codereview */
 57 EOF
```

```
**********************************************************
    1441 Fri Aug 31 05:08:59 2012
new/grubadm/error.c
menuadm->grubadm, ba_path->module, various changes, starting adding serial termi
**********************************************************
```

```
/*
 * 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 2012, Daniil Lunev. All rights reserved.
 */
#include <stdarg.h>
#include <stdio.h>
#include <stdlib.h>
#include "grubadm.h"
#include "error.h"

char * DEBUG;

void
debug_print (const char * fmt, ...)
{
        va_list args;
        if (DEBUG) {
                va_start (args, fmt);
                vfprintf (stderr, fmt, args);
                va_end (args);
        }
}

void
print_system_error ()
{
        perror(APP_NAME);
}

void
print_error (const char * fmt, ...)
{
        va_list args;
        fprintf (stderr, "%s: ", APP_NAME);
        va_start (args, fmt);
        vfprintf (stderr, fmt, args);
        va_end (args);
        fprintf (stderr, "\n");
}

void
check_debug ()
```



```
{
        DEBUG = getenv("DEBUG");
}
#endif /* ! codereview */
```

```
**********************************************************
    1039 Fri Aug 31 05:08:59 2012
new/grubadm/error.h
menuadm->grubadm, ba_path->module, various changes, starting adding serial termi
**********************************************************
```

```
/*
 * 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 2012, Daniil Lunev. All rights reserved.
 */
extern char * DEBUG;

void
debug_print (const char * fmt, ...);

void
print_system_error ();

void
print_error (const char * fmt, ...);

void
check_debug ();
#endif /* ! codereview */
```

```
**********************************************************
    6974 Fri Aug 31 05:09:00 2012
new/grubadm/grubadm.1m
license + man
**********************************************************
'\" te
.\" Copyright 2012, Daniil Lunev. All rights reserved.
.TH GRUBADM 1M "July 25, 2012"
.SH NAME
grubadm \- GRUB2 Illumos bootmenu manager
.SH SYNOPSIS
.LP
.nf
\fBgrubadm\fR [\fB--clone\fR] [\fB--new\fR] [\fB--delete\fR] [\fB--list\fR]
[\fB\--helpfR] [\fB--all\fR] [\fB--enable-hyper\fR] [\fB--disable-hyper\fR]
[\fB--get-name\fR] [\fB--get-uuid\fR] [\fB--get-pool\fR][\fB--get-dataset\fR]
[\fBget-kernel\fR] [\fB--get-opts\fR] [\fB--get-modules\fR] [\fB--get-default\fR
[\fB--get-timeout\fR] [\fB--get-terminal\fR] [\fB--get-serial\fR] [\fB--get-all\
[\fB--default\fR] [\fB--set-name\fR \fIname\fR] [\fB--set-uuid\fR \fIuuid\fR]
[\fB--set-pool\fR \fIpool\fR] [\fB--set-dataset\fR \fIdataset\fR]
[\fB--set-kernel\fR \fIkernel\fR] [\fB--set-opts\fR \fIoptions\fR]
[\fB--set-module\fR \fImodule\fR] [\fB--set-default\fR \fIdefault entry numder\f
[\fB--set-timeout\fR \fItimeout\fR] [\fB--set-terminal\fR \fIterminals\fR]
[\fB--set-serial\fR \fIserial\fR] [\fB--name\fR \fIname\fR] [\fB--uuid\fR \fIuui
[\fB--pool\fR \fIpool\fR] [\fB--dataset\fR \fIdataset\fR] [\fB--kernel\fR \fIker
[\fB--module\fR \fImodule\fR] [\fB--number\fR \fInumber\fR]
[\fB--alt-root\fR \fIalt-root\fR] [\fB--clear\fR]
.fi

.SH DESCRIPTION
.sp
.LP
The command \fBgrubadm\fR is used to manipulate GRUB2 Illumos entries and GRUB2
booting parameters. It modify only \fBillumos.cfg\fR and don't influence on entr
that placed in different locations. If you want to see illumos entries in
the GRUB2 boot menu, be sure that \fBgrub.cfg\fR contain line "illumos_entries
/@/boot/illumos.cfg".
.SH OPTIONS
.LP
The following options are supported:
.sp
.ne 2
.na
\fB\fB--list\fR\fR
.ad
.sp .6
.RS 4n
List names of all entries and global parameters values
.RE

.sp
.ne 2
.na
\fB\fB--delete\fR\fR
.ad
.sp .6
.RS 4n
Delete entry (don't stack with --clone, --new, --get-*, --set-*,
--enable-hyper, --disable-hyper)
.RE

.sp
.ne 2
.na
\fB\fB--new\fR\fR
.ad
```



```
.sp .6
.RS 4n
Add new entry (don't stack with --clone, --delete, --all)
.RE

.sp
.ne 2
.na
\fB\fB--default\fR\fR
.ad
.sp .6
.RS 4n
Fill entry with default values
.RE

.sp
.ne 2
.na
\fB\fB--clone\fR\fR
.ad
.sp .6
.RS 4n
Clone existing entry (don't stack with --delete, --new, --all)
.RE

.sp
.ne 2
.na
\fB\fB--clear\fR\fR
.ad
.sp .6
.RS 4n
Delete all entries
.RE

.sp
.ne 2
.na
\fB\fB--help\fR\fR
.ad
.sp .6
.RS 4n
Show usage
.RE

.sp
.ne 2
.na
\fB\fB--all\fR\fR
.ad
.sp .6
.RS 4n
Apply action to all appropriate entries
.RE

.sp
.ne 2
.na
\fB\fB--number\fR\fR \fInumber\fR
.ad
.sp .6
.RS 4n
Get entry by number. If an unappropriate value are passed,
default entry will be getted.
.RE
```

```
.sp
.ne 2
.na
\fB\fB--name\fR\fR \fIname\fR
.ad
.sp .6
.RS 4n
Find entry by name
.RE

.sp
.ne 2
.na
\fB\fB--uuid\fR\fR \fIuuid\fR
.ad
.sp .6
.RS 4n
Find entry by pool uuid
.RE

.sp
.ne 2
.na
\fB\fB--pool\fR\fR \fIpool\fR
.ad
.sp .6
.RS 4n
Find entry by pool name
.RE

.sp
.ne 2
.na
\fB\fB--dataset\fR\fR \fIdataset\fR
.ad
.sp .6
.RS 4n
Find entry by dataset
.RE

.sp
.ne 2
.na
\fB\fB--kernel\fR\fR \fIkernel\fR
.ad
.sp .6
.RS 4n
Find entry by kernel
.RE

.sp
.ne 2
.na
\fB\fB--module\fR\fR \fImodule\fR
.ad
.sp .6
.RS 4n
Find entry by module
.RE

.sp
.ne 2
.na
\fB\fB--get-name\fR\fR
.ad
.sp .6
```



```
.RS 4n
Get entry name
.RE

.sp
.ne 2
.na
\fB\fB--get-pool\fR\fR
.ad
.sp .6
.RS 4n
Get entry pool name
.RE

.sp
.ne 2
.na
\fB\fB--get-uuid\fR\fR
.ad
.sp .6
.RS 4n
Get entry pool uuid
.RE

.sp
.ne 2
.na
\fB\fB--get-dataset\fR\fR
.ad
.sp .6
.RS 4n
Get entry dataset
.RE

.sp
.ne 2
.na
\fB\fB--get-kernel\fR\fR
.ad
.sp .6
.RS 4n
Get entry kernel line
.RE

.sp
.ne 2
.na
\fB\fB--get-opts\fR\fR
.ad
.sp .6
.RS 4n
Get entry kernele opts
.RE

.sp
.ne 2
.na
\fB\fB--get-modules\fR\fR
.ad
.sp .6
.RS 4n
Get entry modules
.RE

.sp
.ne 2
```

```
.na
\fB\fB--get-default\fR\fR
.ad
.sp .6
.RS 4n
Get default entry number
.RE

.sp
.ne 2
.na
\fB\fB--get-timeout\fR\fR
.ad
.sp .6
.RS 4n
Get timeout
.RE

.sp
.ne 2
.na
\fB\fB--get-serial\fR\fR
.ad
.sp .6
.RS 4n
Get serial port configureation info
.RE

.sp
.ne 2
.na
\fB\fB--get-terminal\fR\fR
.ad
.sp .6
.RS 4n
Get terminal
.RE

.sp
.ne 2
.na
\fB\fB--get-all\fR\fR
.ad
.sp .6
.RS 4n
Get all information about appropriate entries
.RE

.sp
.ne 2
.na
\fB\fB--set-name\fR\fR \fIname\fR
.ad
.sp .6
.RS 4n
Set entry name
.RE

.sp
.ne 2
.na
\fB\fB--set-pool\fR\fR \fIpool\fR
.ad
.sp .6
.RS 4n
Set entry pool name
```



```
.RE

.sp
.ne 2
.na
\fB\fB--set-uuid\fR\fR \fIuuid\fR
.ad
.sp .6
.RS 4n
Set entry pool uuid
.RE

.sp
.ne 2
.na
\fB\fB--set-dataset\fR\fR \fIdataset\fR
.ad
.sp .6
.RS 4n
Set entry dataset
.RE

.sp
.ne 2
.na
\fB\fB--set-kernel\fR\fR \fIkernel\fR
.ad
.sp .6
.RS 4n
Set entry kernel line
.RE

.sp
.ne 2
.na
\fB\fB--set-opts\fR\fR \fIoptions\fR
.ad
.sp .6
.RS 4n
Set entry kernele opts
.RE

.sp
.ne 2
.na
\fB\fB--set-module\fR\fR \fImodule\fR
.ad
.sp .6
.RS 4n
Set entry module (all earlier specified modules will be deleted,
if you want to specify multiple modules, use several --set-module
in a single command)
.RE

.sp
.ne 2
.na
\fB\fB--set-default\fR\fR \fIdefault entry number\fR
.ad
.sp .6
.RS 4n
Set default entry number
.RE

.sp
.ne 2
```

```
.na
\fB\fB--set-timeout\fR\fR \fItimeout\fR
.ad
.sp .6
.RS 4n
Set timeout
.RE

.sp
.ne 2
.na
\fB\fB--set-serial\fR\fR \fIserial_port_params\fR
.ad
.sp .6
.RS 4n
Set serial port configureation info
.RE

.sp
.ne 2
.na
\fB\fB--set-terminal\fR\fR \fIterminal\fR
.ad
.sp .6
.RS 4n
Set terminal
.RE

.sp
.ne 2
.na
\fB\fB--enable-hyper\fR\fR
.ad
.sp .6
.RS 4n
Convert entry to boot with xen
.RE

.sp
.ne 2
.na
\fB\fB--disable-hyper\fR\fR
.ad
.sp .6
.RS 4n
Convert entry from xen-mode to normal boot mod
.RE

.SH EXAMPLES
.LP
\fBExample 1\fR
.sp
.LP
List all entries

.sp
.in +2
.nf
example% \fBgrubadm --list\fR
.fi
.in -2
.sp

.LP
\fBExample 2\fR
.sp
```



```
.LP
Add new entry with default paramters

.sp
.in +2
.nf
example% \fBgrubadm --new --default --set-name test\fR
.fi
.in -2
.sp

.LP
\fBExample 3\fR
.sp
.LP
Delete all entries with specified dataset

.sp
.in +2
.nf
example% \fBgrubadm --delete --all --dataset "rpool/ROOT/test_pool"\fR
.fi
.in -2
.sp

.LP
\fBExample 4\fR
.sp
.LP
Clone entry and set diffrent kernel options

.sp
.in +2
.nf
example% \fBgrubadm --clone --name test --set-name new_test --set-opt "-s"\fR
.fi
.in -2
.sp

.LP
\fBExample 5\fR
.sp
.LP
Change kernel and convert entry to hypervisor

.sp
.in +2
.nf
example% \fBgrubadm --name test --set-kernel "/platform/i86pc/kernel/unix" --ena
.fi
.in -2
.sp

.LP
\fBExample 6\fR
.sp
.LP
Set up serial console

.sp
.in +2
.nf
example% \fBgrubadm --set-serial "--unit=0 --speed=9600" --set-terminal "serial"
.fi
.in -2
.sp
```

```
525 .LP
526 \fBExample 7\fR
527 .sp
528 .LP
529 Get default entry number

531 .sp
532 .in +2
533 .nf
534 example% \fBgrubadm --get-default\fR
535 .fi
536 .in -2
537 .sp

539 .SH FILES
540 .sp
541 .ne 2
542 .na
543 /pool/boot/illumos.cfg
544 .ad
545 .sp .6
546 .RS 4n
547 Bootmenu file.
548 .RE

550 .SH NOTES
551 grubadm is replacement of bootadm boot menu functionality for grub2. You can't
552 use bootadm menu functionality on with GRUB2 as well as you can't use grubadm
553 with GRUB-legacy.

555 #endif /* ! codereview */
```

```
**********************************************************
   13773 Fri Aug 31 05:09:00 2012
new/grubadm/grubadm.c
rename
**********************************************************
```

```
/*
 * 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 2012, Daniil Lunev. All rights reserved.
 */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <getopt.h>
#include <assert.h>
#include "grubadm.h"
#include "error.h"
#include "menu.h"

menu_list get_arg = { 0 };
menu_list find_arg = { 0 };
menu_list set_arg = { 0 };
int get_timeout = 0;
int get_default = 0;
int get_serial = 0;
int get_terminal = 0;
int lst_flag = 0;
int new_flag = 0;
int all_flag = 0;
int del_flag = 0;
int fnd_flag = 0;
int hlp_flag = 0;
int get_flag = 0;
int set_flag = 0;
int cln_flag = 0;
int clr_flag = 0;
int enh_flag = 0;
int dih_flag = 0;
int num_flag = 0;
int number;
char * set_timeout = NULL;
char * set_default = NULL;
char * set_serial = NULL;
char * set_terminal = NULL;
char * alt_root = NULL;

static char opt_string[] = "landh?R:";
```



```
static const struct option options[] = {
        { "list", no_argument, NULL, 'l' },
        { "all", no_argument, NULL, 'a' },
        { "new", no_argument, NULL, 'n' },
        { "delete", no_argument, NULL, 'd' },
        { "help", no_argument, NULL, 'h' },
        { "name", required_argument, NULL, 4 },
        { "pool", required_argument, NULL, 5 },
        { "uuid", required_argument, NULL, 6 },
        { "dataset", required_argument, NULL, 7 },
        { "kernel", required_argument, NULL, 8 },
        { "module", required_argument, NULL, 9 },
        { "set-name", required_argument, NULL, 10 },
        { "set-pool", required_argument, NULL, 11 },
        { "set-uuid", required_argument, NULL, 12 },
        { "set-dataset", required_argument, NULL, 13 },
        { "set-kernel", required_argument, NULL, 14 },
        { "set-opts", required_argument, NULL, 15 },
        { "set-module", required_argument, NULL, 16 },
        { "set-default", required_argument, NULL, 17 },
        { "set-timeout", required_argument, NULL, 18 },
        { "get-name", no_argument, NULL, 19 },
        { "get-pool", no_argument, NULL, 20 },
        { "get-uuid", no_argument, NULL, 21 },
        { "get-dataset", no_argument, NULL, 22 },
        { "get-kernel", no_argument, NULL, 23 },
        { "get-opts", no_argument, NULL, 24 },
        { "get-modules", no_argument, NULL, 25 },
        { "get-default", no_argument, NULL, 26 },
        { "get-timeout", no_argument, NULL, 27 },
        { "get-all", no_argument, NULL, 28 },
        { "alt-root", required_argument, NULL, 'R' },
        { "default", no_argument, NULL, 30 },
        { "set-serial", required_argument, NULL, 31 },
        { "set-terminal", required_argument, NULL, 32 },
        { "get-serial", no_argument, NULL, 33 },
        { "get-terminal", no_argument, NULL, 34 },
        { "clone", no_argument, NULL, 35 },
        { "enable-hyper", no_argument, NULL, 36 },
        { "disable-hyper", no_argument, NULL, 37 },
        { "clear", no_argument, NULL, 38 },
        { "number", required_argument, NULL, 39 },
        { NULL, no_argument, NULL, 0 },
};

static void
usage ()
{
        printf (
                "Usage:\n"
                "--list list all entries\n"
                "--all apply actions to all appropriate entries\n"
                "--new create new entry\n"
                "--delete delete entry\n"
                "--clone clone existing entry\n"
                "--default set default entry properties\n"
                "--help show this message\n"
                "--number <opt> find entry by number (-1 = default entry)\n"
                "--name <opt> find entry by name\n"
                "--pool <opt> by pool name\n"
                "--uuid <opt> by uuid\n"
                "--dataset <opt> by dataset\n"
                "--kernel <opt> by kernel\n"
                "--module <opt> by module\n"
                "--set-name <opt> set new name to entry\n"
                "--set-pool <opt>\n"
```

```
128                 "--set-uuid <opt>\n"
129                 "--set-kernel <opt>\n"
130                 "--set-opts <opt>\n"
131                 "--set-module <opt>\n"
132                 "--set-default <opt>\n"
133                 "--set-timeout <opt>\n"
134                 "--get-name get entry name\n"
135                 "--get-pool\n"
136                 "--get-uuid\n"
137                 "--get-kernel\n"
138                 "--get-opts\n"
139                 "--get-module\n"
140                 "--get-default\n"
141                 "--get-timeout\n"
142                 "--get-all retrieve all params of entry\n"
143                 "--alt-root <opt> alternate root directory\n"
144                 "--default set default fields value to entry\n"
145                 "--set-serial set params of serial port\n"
146                 "--set-terminal set default terminal\n"
147                 "--get-serial get params of serial port"
148                 "--get-terminal get default terminal\n"
149                 "--enable-hyper convert entry to boot with xen\n"
150                 "--disable-hyper convert entry to normal boot\n"
151                 "--clear delete all entries\n"
152         );
153 }

155 static char *
156 get_pool (const char * file)
157 {
158         char buf[MAX_STRING_SIZE];
159         char * tmp;
160         FILE * pipe;

162         const char fnc[] = "get_pool";

164         debug_print ("%s\narg: %s\n", fnc, file);

166         strcpy (buf, "/usr/bin/df -h ");
167         strcat (buf, file);
168
169         pipe = popen (buf, "r");
170         if (! pipe)
171                 return NULL;

173         fgets (buf, MAX_STRING_SIZE, pipe);
174         if (! fgets (buf, MAX_STRING_SIZE, pipe)) {
175                 pclose (pipe);
176                 return NULL;
177         }

179         pclose (pipe);
180         tmp = strchr (buf, '/');
181         *tmp = 0;
182         tmp = strdup (buf);
183         return tmp;
184 }

186 static int
187 parse_args (int argc, char ** argv)
188 {
189         int opt = 0;
190         int index;
191
192         const char fnc[] = "parse_args";
```



```
194         debug_print ("%s\n", fnc);

196         while ((opt = getopt_long (argc, argv, opt_string, options, &index)) !=
197                 switch (opt) {
198                 case 'a':
199                         ++all_flag;
200                         break;
201                 case 'd':
202                         ++del_flag;
203                         break;
204                 case 'l':
205                         ++lst_flag;
206                         break;
207                 case 'n':
208                         ++new_flag;
209                         break;
210                 case 'h':
211                 case '?':
212                         ++hlp_flag;
213                         break;
214                 case 4:
215                         ++fnd_flag;
216                         strcpy (find_arg.entry.entry_name, optarg);
217                         break;
218                 case 5:
219                         ++fnd_flag;
220                         strcpy (find_arg.entry.pool_label, optarg);
221                         break;
222                 case 6:
223                         ++fnd_flag;
224                         strcpy (find_arg.entry.pool_uuid, optarg);
225                         break;
226                 case 7:
227                         ++fnd_flag;
228                         strcpy (find_arg.entry.dataset, optarg);
229                         break;
230                 case 8:
231                         ++fnd_flag;
232                         strcpy (find_arg.entry.kernel, optarg);
233                         break;
234                 case 9:
235                         ++fnd_flag;
236                         find_arg.entry.modules[(find_arg.entry.modules_amount)++
237                         break;
238                 case 10:
239                         ++set_flag;
240                         strcpy (set_arg.entry.entry_name, optarg);
241                         break;
242                 case 11:
243                         ++set_flag;
244                         strcpy (set_arg.entry.pool_label, optarg);
245                         break;
246                 case 12:
247                         ++set_flag;
248                         strcpy (set_arg.entry.pool_uuid, optarg);
249                         break;
250                 case 13:
251                         ++set_flag;
252                         strcpy (set_arg.entry.dataset, optarg);
253                         break;
254                 case 14:
255                         ++set_flag;
256                         strcpy (set_arg.entry.kernel, optarg);
257                         break;
258                 case 15:
259                         ++set_flag;
```

```
260                         strcpy (set_arg.entry.kernel_opts, optarg);
261                         break;
262                 case 16:
263                         ++set_flag;
264                         set_arg.entry.modules[(set_arg.entry.modules_amount)++]
265                         break;
266                 case 17:
267                         set_default = strdup (optarg);
268                         break;
269                 case 18:
270                         set_timeout = strdup (optarg);
271                         break;
272                 case 19:
273                         ++get_flag;
274                         get_arg.entry.entry_name[0] = 1;
275                         break;
276                 case 20:
277                         ++get_flag;
278                         get_arg.entry.pool_label[0] = 1;
279                         break;
280                 case 21:
281                         ++get_flag;
282                         get_arg.entry.pool_uuid[0] = 1;
283                         break;
284                 case 22:
285                         ++get_flag;
286                         get_arg.entry.dataset[0] = 1;
287                         break;
288                 case 23:
289                         ++get_flag;
290                         get_arg.entry.kernel[0] = 1;
291                         break;
292                 case 24:
293                         ++get_flag;
294                         get_arg.entry.kernel_opts[0] = 1;
295                         break;
296                 case 25:
297                         ++get_flag;
298                         get_arg.entry.modules = (char**) 1;
299                         break;
300                 case 26:
301                         ++get_default;
302                         break;
303                 case 27:
304                         ++get_timeout;
305                         break;
306                 case 28:
307                         ++get_flag;
308                         memset (&(get_arg.entry), 0xFF, sizeof (menu_entry));
309                         break;
310                 case 'R':
311                         alt_root = strdup (optarg);
312                         break;
313                 case 30:
314                         ++set_flag;
315                         strcpy (set_arg.entry.entry_name, "Menuadm default");
316                         strcpy (set_arg.entry.pool_label, get_pool ("/"));
317                         strcpy (set_arg.entry.kernel, "/platform/i86pc/kernel/$I
318                         strcpy (set_arg.entry.kernel_opts, "-B $ZFS_BOOTFS");
319                         set_arg.entry.modules[0] = strdup ("/platform/i86pc/$ISA
320                         set_arg.entry.modules_amount = 1;
321                         break;
322                 case 31:
323                         set_serial = strdup (optarg);
324                         break;
325                 case 32:
```



```
326                         set_terminal = strdup (optarg);
327                         break;
328                 case 33:
329                         ++get_serial;
330                         break;
331                 case 34:
332                         ++get_terminal;
333                         break;
334                 case 35:
335                         ++cln_flag;
336                         break;
337                 case 36:
338                         ++enh_flag;
339                         break;
340                 case 37:
341                         ++dih_flag;
342                         break;
343                 case 38:
344                         ++clr_flag;
345                         break;
346                 case 39:
347                         ++num_flag;
348                         sscanf (optarg, "%d", &number);
349                         break;
350                 default:
351                         return 0;
352                 }
353         }
354         return 1;
355 }

357 static int
358 check_flags ()
359 {
360         const char fnc[] = "check_flags";

362         debug_print ("%s\n", fnc);

364         if (new_flag)
365                 if (del_flag || all_flag || cln_flag || fnd_flag || num_flag)
366                         return 0;

368         if (del_flag)
369                 if ((! fnd_flag && ! num_flag) || set_flag || get_flag || cln_fl
370                         return 0;

372         if (cln_flag)
373                 if (! fnd_flag || all_flag)
374                         return 0;

376         if (fnd_flag)
377                 if (num_flag)
378                         return 0;
379         return 1;
380 }

382 static void
383 apply_set (menu_list * entry, menu_list * set)
384 {
385         unsigned int i = 0;

387         const char fnc[] = "apply_set";

389         debug_print ("%s\n", fnc);

391         if (set->entry.entry_name[0])
```

```
392                 strcpy (entry->entry.entry_name, set->entry.entry_name);

394         if (set->entry.pool_label[0])
395                 strcpy (entry->entry.pool_label, set->entry.pool_label);

397         if (set->entry.pool_uuid[0])
398                 strcpy (entry->entry.pool_uuid, set->entry.pool_uuid);

400         if (set->entry.dataset[0])
401                 strcpy (entry->entry.dataset, set->entry.dataset);

403         if (set->entry.kernel[0])
404                 strcpy (entry->entry.kernel, set->entry.kernel);

406         if (set->entry.kernel_opts[0])
407                 strcpy (entry->entry.kernel_opts, set->entry.kernel_opts);

409         if (set->entry.modules_amount) {
410                 for (i = 0; i < entry->entry.modules_amount; ++i)
411                         free (entry->entry.modules[i]);

413                 for (i = 0; i < set->entry.modules_amount; ++i)
414                         entry->entry.modules[i] = strdup (set->entry.modules[i])

416                 entry->entry.modules_amount = set->entry.modules_amount;
417         }
418 }

420 static void
421 output_entry (menu_list * entry, menu_list * get)
422 {
423         unsigned int i = 0;

425         const char fnc[] = "output_entry";

427         debug_print ("%s\n", fnc);

429         if (get->entry.entry_name[0])
430                 printf("%s\n", entry->entry.entry_name);

432         if (get->entry.pool_label[0])
433                 printf("%s\n", entry->entry.pool_label);

435         if (get->entry.pool_uuid[0])
436                 printf("%s\n", entry->entry.pool_uuid);

438         if (get->entry.dataset[0])
439                 printf("%s\n", entry->entry.dataset);

441         if (get->entry.kernel[0])
442                 printf("%s\n", entry->entry.kernel);

444         if (get->entry.kernel_opts[0])
445                 printf("%s\n", entry->entry.kernel_opts);

447         if (get->entry.modules)
448                 for (i = 0; i < entry->entry.modules_amount; ++i)
449                         printf("%s\n", entry->entry.modules[i]);

451         printf("----------------------------------------\n");
452 }

454 int
455 main (int argc, char ** argv)
456 {
457         char * pool;
```



```
458         char config[MAX_STRING_SIZE];
459         char tmp_config[MAX_STRING_SIZE];
460         menu_list * menu;
461         menu_list * tmp;
462         menu_list * entry;
463         int entries_amount;
464         int d_num;

466         check_debug ();

468         if (argc < 2)
469                 ++lst_flag;

471         set_arg.entry.modules = (char **) calloc (MAX_MODULE_AMOUNT, sizeof (cha
472         find_arg.entry.modules = (char **) calloc (MAX_MODULE_AMOUNT, sizeof (ch
473         if (! parse_args (argc, argv))
474                 return 1;

476         if (hlp_flag) {
477                 usage ();
478                 return 0;
479         }

481         if (! check_flags ()) {
482                 print_error ("wrong option set, check \"man grubadm\"");
483                 return 1;
484         }

486         pool = get_pool (alt_root ?: "/");
487         strcpy (config, "/");
488         if (alt_root)
489                 strcat (config, alt_root);
490         strcat (config, "/");
491         strcat (config, pool);
492         strcat (config, "/");
493         strcpy (tmp_config, config);
494         strcat (config, CONFIG_PATH);
495         strcat (tmp_config, TMP_CONFIG_PATH);

497         menu = parse_file (config, &entries_amount);
498         if (entries_amount == -1) {
499                 return 1;
500         }

502         tmp = menu;
503         entry = NULL;

505         do {
506                 if (clr_flag) {
507                         menu = NULL;
508                         break;
509                 }

511                 if (set_default) {
512                         sscanf (set_default, "%d", &d_num);
513                         if (d_num < entries_amount)
514                                 strcpy (default_entry, set_default);
515                 }
516
517                 if (set_timeout)
518                         strcpy (timeout, set_timeout);

520                 if (set_serial)
521                         strcpy (serial, set_serial);

523                 if (set_terminal)
```

```
524                         strcpy (terminal, set_terminal);

526                 if (fnd_flag) {
527                         entry = find_entry (tmp, &find_arg);
528                         if (! entry) {
529                                 print_error (MSG_ERR_NO_ENTRY);
530                                 break;
531                         }
532                 }

534                 if (num_flag) {
535                         if ((number < 0) || (number >= entries_amount)) {
536                                 if (! default_entry[0]) {
537                                         number = 0;
538                                 } else {
539                                         sscanf (default_entry, "%d", &number);
540                                 }
541                         }
542                         if (number >= entries_amount)
543                                 number = 0;
544                         entry = get_entry_by_number (menu, number);
545                         if (! entry) {
546                                 print_error (MSG_ERR_NO_ENTRY);
547                                 break;
548                         }
549                 }

551                 if (cln_flag) {
552                         entry = clone_entry (entry);
553                         if (! entry) {
554                                 print_error (MSG_ERR_NO_ENTRY);
555                                 break;
556                         }
557                 }

559                 if (set_flag) {
560                         if (entry)
561                                 apply_set (entry, &set_arg);
562                         else if (new_flag)
563                                 add_entry (&menu, &set_arg);
564                         else
565                                 return 1;
566                 }

568                 if (enh_flag)
569                         if (! enable_hyper (entry)) {
570                                 print_error (MSG_ERR_HYPER);
571                                 break;
572                         }
573
574                 if (dih_flag)
575                         if (! disable_hyper (entry)) {
576                                 print_error (MSG_ERR_HYPER);
577                                 break;
578                         }

580                 if (cln_flag)
581                         add_entry (&menu, entry);

583                 if (get_flag)
584                         if (entry)
585                                 output_entry (entry, &get_arg);

587                 if (all_flag)
588                         tmp = entry->next;
589                 else
```



```
590                         tmp = NULL;

592                 if (del_flag)
593                         delete_entry (&menu, entry);
594         } while (tmp);

596         if (get_default)
597                 printf ("%s%s%s\n", cmd_list[CMD_DEFAULT], SEPARATOR, default_en

599         if (get_timeout)
600                 printf ("%s%s%s\n", cmd_list[CMD_TIMEOUT], SEPARATOR, timeout);

602         if (get_serial)
603                 printf ("%s%s%s\n", cmd_list[CMD_SERIAL], SEPARATOR, serial);

605         if (get_terminal)
606                 printf ("%s%s%s\n", cmd_list[CMD_TERMINAL], SEPARATOR, terminal)

608         if (lst_flag) {
609                 list_menu (menu);
610         }

612         write_menu (config, tmp_config, menu);
613         free_menu (&menu);
614         return 0;
615 }
616 #endif /* ! codereview */
```

****
1402 Fri Aug 31 05:09:00 2012
new/grubadm/grubadm.h
rename
****

```
 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 /*
22  * Copyright 2012, Daniil Lunev. All rights reserved.
23  */
24 #define APP_NAME "grubadm"
25 #define CONFIG_PATH "/boot/illumos.cfg"
26 #define TMP_CONFIG_PATH "/boot/illumos.cfg.tmp"

28 #define HYPER_KERNEL "/boot/$ISADIR/xen.gz"
29 #define HYPER_KERNEL_OPTS "console=vga"

31 #define MAX_STRING_SIZE 1024
32 #define MAX_MODULE_AMOUNT 8

34 #define MSG_ERR_LONG_LINE "Too long line"
35 #define MSG_ERR_WRONG_SYNTAX "Syntax error"
36 #define MSG_ERR_SAME_NAME "Duplicate entry definition"
37 #define MSG_ERR_NO_ENTRY "Entry with defined parameters can not be found"
38 #define MSG_ERR_HYPER "Hyper operation failed"
39 #endif /* ! codereview */
```

```
**********************************************************
   13007 Fri Aug 31 05:09:01 2012
new/grubadm/menu.c
menuadm->grubadm, ba_path->module, various changes, starting adding serial termi
**********************************************************
```

```
 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 /*
22  * Copyright 2012, Daniil Lunev. All rights reserved.
23  */
24 #include <stdio.h>
25 #include <stdlib.h>
26 #include <string.h>
27 #include "grubadm.h"
28 #include "error.h"
29 #include "menu.h"

31 char default_entry[MAX_STRING_SIZE] = { 0 };
32 char timeout[MAX_STRING_SIZE] = { 0 };
33 char serial[MAX_STRING_SIZE] = { 0 };
34 char terminal[MAX_STRING_SIZE] = { 0 };

36 const char * cmd_list[] = {
37         "entry_name",
38         "pool_label",
39         "pool_uuid",
40         "data_set",
41         "kernel_path",
42         "kernel_options",
43         "module",
44         "default_entry",
45         "timeout",
46         "serial",
47         "terminal",
48         NULL
49 };

51 void
52 free_menu (menu_list ** menu)
53 {
54         unsigned int i;
55         menu_list * iter = *menu;
56         menu_list * tmp = *menu;

58         const char fnc[] = "free_menu";

60         debug_print ("%s\n", fnc);
```



```
62         while (iter) {
63                 if (iter->entry.modules_amount)
64                         for (i = 0; i < iter->entry.modules_amount; ++i)
65                                 free (iter->entry.modules[i]);
66                 free (iter->entry.modules);
67                 tmp = iter;
68                 iter = iter->next;
69                 free (tmp);
70         }

72         *menu = NULL;
73 }

75 static menu_list *
76 new_entry (menu_list ** menu, const char * entry_name, const unsigned int num)
77 {
78         menu_list * added_entry;
79         menu_list * iter;

81         const char fnc[] = "new_entry";

83         debug_print ("%s\nargs : %s | %u\n", fnc, entry_name, num);

85         added_entry = (menu_list *) calloc (1, sizeof (menu_list));
86         if (! added_entry) {
87                 print_system_error ();
88                 return NULL;
89         }

91         memset ((void *) added_entry, 0, sizeof (menu_list));
92         added_entry->entry_number = num;
93         strcpy (added_entry->entry.entry_name, entry_name);
94         added_entry->entry.modules =
95                 (char **) calloc (MAX_MODULE_AMOUNT, sizeof (char *));

97         if (! added_entry->entry.modules) {
98                 free_menu (&added_entry);
99                 print_system_error ();
100                return NULL;
101        }

103        if (! *menu) {
104                *menu = added_entry;
105        } else {
106                for (iter = *menu; iter->next; iter = iter->next)
107                        if (! strcmp (iter->entry.entry_name, entry_name)) {
108                                print_error ("%s %s", MSG_ERR_SAME_NAME, entry_n
109                                free_menu (&added_entry);
110                                return NULL;
111                        }
112                if (! strcmp (iter->entry.entry_name, entry_name)) {
113                        print_error ("%s %s", MSG_ERR_SAME_NAME, entry_name);
114                        free_menu (&added_entry);
115                        return NULL;
116                }
117                iter->next = added_entry;
118        }

120        return added_entry;
121 }

123 menu_list *
124 add_entry (menu_list ** menu, menu_list * entry)
125 {
126        menu_list * iter;
```

```
128         const char fnc[] = "add_entry";

130         debug_print ("%s\n", fnc);

132         if (! entry)
133                 return NULL;

135         if (! *menu) {
136                 entry->entry_number = 0;
137                 *menu = entry;
138                 return entry;
139         }

141         for (iter = * menu; iter->next; iter = iter->next)
142                 if (! strcmp (iter->entry.entry_name, entry->entry.entry_name))
143                         return NULL;
144
145         if (! strcmp (iter->entry.entry_name, entry->entry.entry_name))
146                 return NULL;

148         iter->next = entry;
149         entry->entry_number = iter->entry_number + 1;
150         return entry;
151 }

153 menu_list *
154 clone_entry (menu_list * entry)
155 {
156         unsigned int i;

158         char fnc[] = "clone_entry";

160         debug_print ("%s\n", fnc);

162         if (! entry)
163                 return NULL;
164         menu_list * clone = (menu_list *) calloc (1, sizeof (menu_list));
165         memcpy ((void *) clone, (void *) entry, sizeof (menu_list));
166         clone->next = NULL;
167         clone->entry.modules = (char **) calloc (MAX_MODULE_AMOUNT, sizeof (char
168         for (i = 0; i < clone->entry.modules_amount; ++i)
169                 clone->entry.modules[i] = strdup (entry->entry.modules[i]);
170         return clone;
171 }

173 menu_list *
174 enable_hyper (menu_list * entry)
175 {
176         unsigned int i;
177         char kernel_mod[MAX_STRING_SIZE];

179         char fnc[] = "enable_hyper";

181         debug_print ("%s\n", fnc);

183         if (! entry)
184                 return NULL;
185
186         strcpy (kernel_mod, entry->entry.kernel);
187         strcat (kernel_mod, " ");
188         strcat (kernel_mod, entry->entry.kernel);
189         strcat (kernel_mod, " ");
190         strcat (kernel_mod, entry->entry.kernel_opts);

192         strcpy (entry->entry.kernel, HYPER_KERNEL);
193         strcpy (entry->entry.kernel_opts, HYPER_KERNEL_OPTS);
```



```
194
195         for (i = entry->entry.modules_amount; i > 0; --i)
196                 entry->entry.modules[i] = entry->entry.modules[i-1];
197         entry->entry.modules[0] = strdup (kernel_mod);
198         ++(entry->entry.modules_amount);

200         return entry;
201 }

203 menu_list *
204 disable_hyper (menu_list * entry)
205 {
206         unsigned int i;
207         char * tmp;

209         char fnc[] = "disable_hyper";

211         debug_print ("%s\n", fnc);

213         if (! entry)
214                 return NULL;

216         tmp = strchr (entry->entry.modules[0], ' ');
217         if (! tmp)
218                 return NULL;

220         *tmp++ = '\0';

222         strcpy (entry->entry.kernel, entry->entry.modules[0]);
223
224         entry->entry.kernel_opts[0] = '\0';
225         tmp = strchr(tmp, ' ');
226         if (tmp && *(++tmp)) {
227                 strcpy (entry->entry.kernel_opts, tmp);
228         }
229         free (entry->entry.modules[0]);
230         --(entry->entry.modules_amount);

232         for (i = 0; i < entry->entry.modules_amount; ++i)
233                 entry->entry.modules[i] = entry->entry.modules[i + 1];

235         return entry;
236 }

238 int
239 delete_entry (menu_list ** menu, menu_list * del)
240 {
241         menu_list * iter = *menu;
242         int d_num, i_num;

244         const char fnc[] = "delete_entry";

246         debug_print ("%s\n", fnc);

248         if (! menu || ! del)
249                 return 0;

251         i_num = del->entry_number;

253         if (iter == del) {
254                 *menu = (*menu)->next;
255                 iter->next = NULL;
256                 free_menu (&iter);
257         } else {
258                 while (iter->next) {
259                         if (iter->next == del) {
```

```
260                                 iter->next = del->next;
261                                 del->next = NULL;
262                                 free_menu (&del);
263                                 break;
264                         }
265                         iter = iter->next;
266                 }
267         }

269         sscanf (default_entry, "%d", &d_num);

271         if (i_num <= d_num) {
272                 d_num = d_num ? d_num - 1 : 0;
273                 sprintf (default_entry, "%d", d_num);
274         }

276         return 0;
277 }

279 static menu_list *
280 add_param (menu_list ** menu,
281                 const cmd_id id,
282                 const char * value,
283                 unsigned int * num)
284 {
285         menu_list * iter;
286         char * tmp;

288         const char fnc[] = "add_param";

290         debug_print ("%s\narg: %u | %s | %u\n", fnc, (unsigned int) id, value, *

292         if (id == CMD_ENTRY_NAME)
293                 return new_entry (menu, value, (*num)++);

295         if (! *menu) {
296                 print_error ("%s, %s", MSG_ERR_WRONG_SYNTAX, "entry isn't define
297         }

299         for (iter = * menu; iter->next; iter = iter->next)
300                 ;

302         switch (id) {
303         case CMD_POOL_LABEL:
304                 strcpy (iter->entry.pool_label, value);
305                 break;
306         case CMD_POOL_UUID:
307                 strcpy (iter->entry.pool_uuid, value);
308                 break;
309         case CMD_DATA_SET:
310                 strcpy (iter->entry.dataset, value);
311                 break;
312         case CMD_KERNEL_PATH:
313                 strcpy (iter->entry.kernel, value);
314                 break;
315         case CMD_KERNEL_OPTIONS:
316                 strcpy (iter->entry.kernel_opts, value);
317                 break;
318         case CMD_BA_PATH:
319                 if (iter->entry.modules_amount >= MAX_MODULE_AMOUNT)
320                         return NULL;
321                 iter->entry.modules[iter->entry.modules_amount] =
322                         (char *) malloc (MAX_STRING_SIZE);
323                 if (! iter->entry.modules[iter->entry.modules_amount])
324                         return NULL;
325                 strcpy (iter->entry.modules[iter->entry.modules_amount], value);
```



```
326                 ++(iter->entry.modules_amount);
327                 break;
328         default:
329                 return NULL;
330         }
331         return iter;
332 }

334 void
335 list_menu (menu_list * menu)
336 {
337         const char fnc[] = "list_menu";

339         debug_print ("%s\n", fnc);

341         if (*default_entry)
342                 printf ("%s%s%s\n", cmd_list[CMD_DEFAULT], SEPARATOR, default_en

344         if (*timeout)
345                 printf ("%s%s%s\n", cmd_list[CMD_TIMEOUT], SEPARATOR, timeout);

347         if (*serial)
348                 printf ("%s%s%s\n", cmd_list[CMD_SERIAL], SEPARATOR, serial);

350         if (*terminal)
351                 printf ("%s%s%s\n", cmd_list[CMD_TERMINAL], SEPARATOR, terminal)

353         while (menu) {
354                 printf("%u : %s\n", menu->entry_number, menu->entry.entry_name);
355                 menu = menu->next;
356         }
357 }

359 menu_list *
360 find_entry (menu_list * menu, menu_list * find)
361 {
362         unsigned int i = 0;
363         const char fnc[] = "find_entry";

365         debug_print ("%s\n", fnc);

367         if (! menu || ! find)
368                 return NULL;

370         while (menu) {
371                 if (find->entry.entry_name[0])
372                         if (strcmp (menu->entry.entry_name, find->entry.entry_na
373                                 goto out;
374                 if (find->entry.pool_label[0])
375                         if (strcmp (menu->entry.pool_label, find->entry.pool_lab
376                                 goto out;
377                 if (find->entry.pool_uuid[0])
378                         if (strcmp (menu->entry.pool_uuid, find->entry.pool_uuid
379                                 goto out;
380                 if (find->entry.dataset[0])
381                         if (strcmp (menu->entry.dataset, find->entry.dataset))
382                                 goto out;
383                 if (find->entry.kernel[0])
384                         if (strcmp (menu->entry.kernel, find->entry.kernel))
385                                 goto out;

387                 if (find->entry.modules_amount) {
388                         for (i = 0; i < menu->entry.modules_amount; ++i)
389                                 if (! strcmp (menu->entry.modules[i], find->entr
390                                         return menu;
391                         goto out;
```

```
392                 }

394                 return menu;
395 out:
396                 menu = menu->next;
397         }

399         return NULL;
400 }

402 menu_list *
403 get_entry_by_number (menu_list * menu, int number)
404 {
405         while (menu)
406                 if (menu->entry_number == number)
407                         return menu;
408                 else
409                         menu = menu->next;
410         return NULL;
411 }

413 static size_t
414 get_line (FILE * menu_file, char * line)
415 {
416         const char fnc[] = "get_line";
417
418         debug_print ("%s\n", fnc);

420         if (! fgets (line, MAX_STRING_SIZE, menu_file)) {
421                 if (! feof (menu_file))
422                         print_system_error ();
423                 return -1;
424         }

426         if ((line[strlen (line) - 1] != '\n') && (! feof (menu_file))) {
427                 print_error ("%s (> %d)", MSG_ERR_LONG_LINE, MAX_STRING_SIZE);
428                 return -1;
429         }

431         line[strlen (line) - 1] = '\0';

433         return strlen (line);
434 }

436 static cmd_id
437 get_param_id (const char * param)
438 {
439         unsigned int id;

441         const char fnc[] = "get_param_id";

443         debug_print ("%s\narg : %s\n", fnc, param);

445         for (id = 0; cmd_list[id]; ++id)
446                 if (! strcmp (cmd_list[id], param))
447                         break;

449         return id;
450 }

452 menu_list *
453 parse_file (const char * file_name, int * entries_amount)
454 {
455         FILE * menu_file = NULL;
456         unsigned int counter = 0;
457         menu_list * menu = NULL;
```



```
458         char line[MAX_STRING_SIZE];
459         char * param;
460         char * value;
461         cmd_id id;
462
463         const char fnc[] = "parse_file";

465         debug_print ("%s\narg : %s\n", fnc, file_name);

467         menu_file = fopen (file_name, "r");
468         if (! menu_file) {
469                 print_system_error ();
470                 goto out;
471         }

473         for (;;) {
474                 if (get_line (menu_file, line) == -1) {
475                         if (feof (menu_file))
476                                 break;
477                         free_menu (&menu);
478                         counter = -1;
479                         goto out;
480                 }

482                 if (line[0] == '#')
483                         continue;
484
485                 param = line;
486                 while ((*param == ' ') || (*param == '\t'))
487                         ++ param;
488                 if (*param == '\0')
489                         continue;

491                 value = strchr (line, '=');
492                 if (! value) {
493                         print_error ("%s at line '%s'", MSG_ERR_WRONG_SYNTAX, li
494                         free_menu (&menu);
495                         counter = -1;
496                         goto out;
497                 }
498                 *value++ = '\0';

500                 id = get_param_id (param);
501                 if (id == CMD_UNKNOWN) {
502                         print_error ("%s at line '%s'", MSG_ERR_WRONG_SYNTAX, li
503                         free_menu (&menu);
504                         counter = -1;
505                         goto out;
506                 }

508                 if (id == CMD_DEFAULT) {
509                         strcpy (default_entry, value);
510                 } else if (id == CMD_TIMEOUT) {
511                         strcpy (timeout, value);
512                 } else if (id == CMD_SERIAL) {
513                         strcpy (serial, value);
514                 } else if (id == CMD_TERMINAL) {
515                         strcpy (terminal, value);
516                 } else if (! add_param (&menu, id, value, &counter)) {
517                         free_menu (&menu);
518                         counter = -1;
519                         goto out;
520                 }
521         }

523 out:
```

```
524         if (menu_file)
525                 fclose (menu_file);

527         *entries_amount = counter;

529         return menu;
530 }

532 int
533 write_menu (const char * config, const char * tmp_config, menu_list * menu)
534 {
535         FILE * tmp_menu = fopen (tmp_config, "w");
536         unsigned int i = 0;

538         const char fnc[] = "write_menu";

540         debug_print ("%s\n", fnc);

542         if (! tmp_menu) {
543                 print_system_error();
544                 return 0;
545         }
546
547         if (default_entry[0])
548                 fprintf (tmp_menu, "%s%s%s\n", cmd_list[CMD_DEFAULT], SEPARATOR,
549         if (timeout[0])
550                 fprintf (tmp_menu, "%s%s%s\n\n", cmd_list[CMD_TIMEOUT], SEPARATO
551         if (serial[0])
552                 fprintf (tmp_menu, "%s%s%s\n\n", cmd_list[CMD_SERIAL], SEPARATOR
553         if (terminal[0])
554                 fprintf (tmp_menu, "%s%s%s\n\n", cmd_list[CMD_TERMINAL], SEPARAT

556         while (menu) {
557                 fprintf (tmp_menu, "%s%s%s\n", cmd_list[CMD_ENTRY_NAME], SEPARAT
558                 if (menu->entry.pool_label[0])
559                         fprintf (tmp_menu, "%s%s%s\n", cmd_list[CMD_POOL_LABEL],
560                 if (menu->entry.pool_uuid[0])
561                         fprintf (tmp_menu, "%s%s%s\n", cmd_list[CMD_POOL_UUID],
562                 if (menu->entry.dataset[0])
563                         fprintf (tmp_menu, "%s%s%s\n", cmd_list[CMD_DATA_SET], S
564                 if (menu->entry.kernel[0])
565                         fprintf (tmp_menu, "%s%s%s\n", cmd_list[CMD_KERNEL_PATH]
566                 if (menu->entry.kernel_opts[0])
567                         fprintf (tmp_menu, "%s%s%s\n", cmd_list[CMD_KERNEL_OPTIO
568                 if (menu->entry.modules_amount)
569                         for (i = 0; i < menu->entry.modules_amount; ++i)
570                                 fprintf (tmp_menu, "%s%s%s\n", cmd_list[CMD_BA_P
571                 fprintf (tmp_menu, "#----------------------------------------
572                 menu = menu->next;
573         }

575         fclose (tmp_menu);

577         remove (config);
578         rename (tmp_config, config);

580         return 1;
581 }
582 #endif /* ! codereview */
```

```
**********************************************************
    2300 Fri Aug 31 05:09:01 2012
new/grubadm/menu.h
menuadm->grubadm, ba_path->module, various changes, starting adding serial termi
**********************************************************
```

```
/*
 * 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 2012, Daniil Lunev. All rights reserved.
 */

#define SEPARATOR "="

typedef struct menu_entry menu_entry;
typedef struct menu_list menu_list;
typedef enum cmd_id cmd_id;

struct menu_entry {
        char entry_name[MAX_STRING_SIZE];
        char pool_label[MAX_STRING_SIZE];
        char pool_uuid[MAX_STRING_SIZE];
        char dataset[MAX_STRING_SIZE];
        char kernel[MAX_STRING_SIZE];
        char kernel_opts[MAX_STRING_SIZE];
        unsigned int modules_amount;
        char ** modules;
};

extern char default_entry[MAX_STRING_SIZE];
extern char timeout[MAX_STRING_SIZE];
extern char serial[MAX_STRING_SIZE];
extern char terminal[MAX_STRING_SIZE];

struct menu_list {
        unsigned int entry_number;
        menu_entry entry;
        menu_list * next;
};

enum cmd_id {
        CMD_ENTRY_NAME,
        CMD_POOL_LABEL,
        CMD_POOL_UUID,
        CMD_DATA_SET,
        CMD_KERNEL_PATH,
        CMD_KERNEL_OPTIONS,
        CMD_BA_PATH,
        CMD_DEFAULT,
```



```
        CMD_TIMEOUT,
        CMD_SERIAL,
        CMD_TERMINAL,
        CMD_UNKNOWN
};

extern const char * cmd_list[];

void
free_menu (menu_list ** menu);

menu_list *
parse_file (const char * file_name, int * entries_amount);

void
list_menu (menu_list * menu);

menu_list *
add_entry (menu_list ** menu, menu_list * entry);

menu_list *
clone_entry (menu_list  *entry);

int
delete_entry (menu_list ** menu, menu_list * del);

menu_list *
find_entry (menu_list * menu, menu_list * find);

menu_list *
enable_hyper (menu_list * entry);

menu_list *
disable_hyper (menu_list * entry);

menu_list *
get_entry_by_number (menu_list * menu, int number);
#endif /* ! codereview */
```

```
**********************************************************
  100804 Fri Aug 31 05:09:01 2012
new/libbe/common/be_utils.c
libbe patch
**********************************************************
______unchanged_portion_omitted_

 347 /*
 348  * Function:    be_append_menu
 349  * Description: Appends an entry for a BE into the menu.lst.
 350  * Parameters:
 351  *              be_name - pointer to name of BE to add boot menu entry for.
 352  *              be_root_pool - pointer to name of pool BE lives in.
 353  *              boot_pool - Used if the pool containing the grub menu is
 354  *                          different than the one contaiing the BE. This
 355  *                          will normally be NULL.
 356  *              be_orig_root_ds - The root dataset for the BE. This is
 357  *                      used to check to see if an entry already exists
 358  *                      for this BE.
 359  *              description - pointer to description of BE to be added in
 360  *                      the title line for this BEs entry.
 361  * Returns:
 362  *              BE_SUCCESS - Success
 363  *              be_errno_t - Failure
 364  * Scope:
 365  *              Semi-private (library wide use only)
 366  */
 367 int
 368 be_append_menu(char *be_name, char *be_root_pool, char *boot_pool,
 369     char *be_orig_root_ds, char *description)
 370 {
 371         zfs_handle_t *zhp = NULL;
 372         char menu_file[MAXPATHLEN];
 373         char be_root_ds[MAXPATHLEN];
 374         char line[BUFSIZ];
 375         char temp_line[BUFSIZ];
 376         char title[MAXPATHLEN];
 377         char *entries[BUFSIZ];
 378         char *tmp_entries[BUFSIZ];
 379         char *pool_mntpnt = NULL;
 380         char *ptmp_mntpnt = NULL;
 381         char *orig_mntpnt = NULL;
 382         boolean_t found_be = B_FALSE;
 383         boolean_t found_orig_be = B_FALSE;
 384         boolean_t found_title = B_FALSE;
 385         boolean_t pool_mounted = B_FALSE;
 386         boolean_t collect_lines = B_FALSE;
 387         FILE *menu_fp = NULL;
 388         int err = 0, ret = BE_SUCCESS;
 389         int i, num_tmp_lines = 0, num_lines = 0;

 391         if (be_name == NULL || be_root_pool == NULL)
 392                 return (BE_ERR_INVAL);

 394         if (boot_pool == NULL)
 395                 boot_pool = be_root_pool;

 397         if ((zhp = zfs_open(g_zfs, be_root_pool, ZFS_TYPE_DATASET)) == NULL) {
 398                 be_print_err(gettext("be_append_menu: failed to open "
 399                     "pool dataset for %s: %s\n"), be_root_pool,
 400                     libzfs_error_description(g_zfs));
 401                 return (zfs_err_to_be_err(g_zfs));
 402         }

 404         /*
 405          * Check to see if the pool's dataset is mounted. If it isn't we'll
```



```
 406          * attempt to mount it.
 407          */
 408         if ((ret = be_mount_pool(zhp, &ptmp_mntpnt, &orig_mntpnt,
 409             &pool_mounted)) != BE_SUCCESS) {
 410                 be_print_err(gettext("be_append_menu: pool dataset "
 411                     "(%s) could not be mounted\n"), be_root_pool);
 412                 ZFS_CLOSE(zhp);
 413                 return (ret);
 414         }

 416         /*
 417          * Get the mountpoint for the root pool dataset.
 418          */
 419         if (!zfs_is_mounted(zhp, &pool_mntpnt)) {
 420                 be_print_err(gettext("be_append_menu: pool "
 421                     "dataset (%s) is not mounted. Can't set "
 422                     "the default BE in the grub menu.\n"), be_root_pool);
 423                 ret = BE_ERR_NO_MENU;
 424                 goto cleanup;
 425         }

 427         /*
 428          * Check to see if this system supports grub
 429          */
 430         if (be_has_grub()) {
 431                 (void) snprintf(menu_file, sizeof (menu_file),
 432                     "%s%s", pool_mntpnt, BE_GRUB_MENU);
 433         } else {
 434                 (void) snprintf(menu_file, sizeof (menu_file),
 435                     "%s%s", pool_mntpnt, BE_SPARC_MENU);
 436         }

 438         be_make_root_ds(be_root_pool, be_name, be_root_ds, sizeof (be_root_ds));

 440         /*
 441          * Iterate through menu first to make sure the BE doesn't already
 442          * have an entry in the menu.
 443          *
 444          * Additionally while iterating through the menu, if we have an
 445          * original root dataset for a BE we're cloning from, we need to keep
 446          * track of that BE's menu entry. We will then use the lines from
 447          * that entry to create the entry for the new BE.
 448          */
 449         if ((ret = be_open_menu(be_root_pool, menu_file,
 450             &menu_fp, "r", B_TRUE)) != BE_SUCCESS) {
 451                 goto cleanup;
 452         } else if (menu_fp == NULL) {
 453                 ret = BE_ERR_NO_MENU;
 454                 goto cleanup;
 455         }

 457         free(pool_mntpnt);
 458         pool_mntpnt = NULL;

 460         while (fgets(line, BUFSIZ, menu_fp)) {
 461                 char *tok = NULL;

 463                 (void) strlcpy(temp_line, line, BUFSIZ);
 464                 tok = strtok(line, "=\n");
 464                 tok = strtok(line, BE_WHITE_SPACE);

 466                 if (tok == NULL || tok[0] == '#') {
 467                         continue;
 468                 } else if (strcmp(tok, "entry_name") == 0) {
 468                 } else if (strcmp(tok, "title") == 0) {
 469                         collect_lines = B_FALSE;
```

```
470                         if ((tok = strtok(NULL, "\n")) == NULL)
471                                 (void) strlcpy(title, "", sizeof (title));
472                         else
473                                 (void) strlcpy(title, tok, sizeof (title));
474                         found_title = B_TRUE;
475
476                         if (num_tmp_lines != 0) {
477                                 for (i = 0; i < num_tmp_lines; i++) {
478                                         free(tmp_entries[i]);
479                                         tmp_entries[i] = NULL;
480                                 }
481                                 num_tmp_lines = 0;
482                         }
483                 } else if (strcmp(tok, "data_set") == 0) {
483                 } else if (strcmp(tok, "bootfs") == 0) {
484                         char *bootfs = strtok(NULL, BE_WHITE_SPACE);
485                         found_title = B_FALSE;
486                         if (bootfs == NULL)
487                                 continue;
488
489                         if (strcmp(bootfs, be_root_ds) == 0) {
490                                 found_be = B_TRUE;
491                                 break;
492                         }
493
494                         if (be_orig_root_ds != NULL &&
495                             strcmp(bootfs, be_orig_root_ds) == 0 &&
496                             !found_orig_be) {
497                                 char str[BUFSIZ];
498                                 found_orig_be = B_TRUE;
499                                 num_lines = 0;
500                                 /*
501                                  * Store the new title line
502                                  */
503                                 (void) snprintf(str, BUFSIZ, "entry_name=%s\n",
503                                 (void) snprintf(str, BUFSIZ, "title %s\n",
504                                     description ? description : be_name);
505                                 entries[num_lines] = strdup(str);
506                                 num_lines++;
507                                 /*
508                                  * If there are any lines between the title
509                                  * and the bootfs line store these. Also
510                                  * free the temporary lines.
511                                  */
512                                 for (i = 0; i < num_tmp_lines; i++) {
513                                         entries[num_lines] = tmp_entries[i];
514                                         tmp_entries[i] = NULL;
515                                         num_lines++;
516                                 }
517
518                                 zprop_get_cbdata_t cb = { 0 };
519                                 zprop_source_t src;
520                                 char * bc = strchr(be_root_ds, '/');
521                                 char sguid[] = "guid";
522
523                                 *bc = '\0';
524                                 cb.cb_first = B_TRUE;
525                                 cb.cb_sources = ZPROP_SRC_ALL;
526                                 cb.cb_type = ZFS_TYPE_POOL;
527                                 zprop_get_list(g_zfs, sguid, &cb.cb_proplist, ZF
528                                 zpool_handle_t * z_hndl = zpool_open(g_zfs, be_r
529                                 *bc = '/';
530                                 if (z_hndl) {
531                                         uint64_t guid = zpool_get_prop_int(z_hnd
532                                         (void) snprintf(str, BUFSIZ, "pool_uuid=
533                                         entries[num_lines] = strdup(str);
```



```
534                                         num_lines++;
535                                         zpool_close(z_hndl);
536                                 }
537 num_tmp_lines = 0;
517                                 num_tmp_lines = 0;
538                                 /*
539                                  * Store the new bootfs line.
540                                  */
541                                 (void) snprintf(str, BUFSIZ, "data_set=%s\n",
521                                 (void) snprintf(str, BUFSIZ, "bootfs %s\n",
542                                     be_root_ds);
543                                 entries[num_lines] = strdup(str);
544                                 num_lines++;
545                                 collect_lines = B_TRUE;
546                         }
547                 } else if (found_orig_be && collect_lines) {
548                         /*
549                          * get the rest of the lines for the original BE and
550                          * store them.
551                          */
552                         if (strstr(line, BE_GRUB_COMMENT) != NULL ||
553                             strstr(line, "BOOTADM") != NULL)
554                                 continue;
555                         if (strcmp(tok, "splashimage") == 0) {
556                                 entries[num_lines] =
557                                     strdup("splashimage "
558                                     "/boot/splashimage.xpm\n");
559                         } else if ((strcmp(tok, "kernel_path") == 0) ||
560                                 (strcmp(tok, "module") == 0))
561                         {
562                                 char * path;
563                                 char * st_path;
564
565                                 st_path = strtok(NULL, "\n");
566                                 path = (char *) malloc(512);
567                                 strcpy(path, tok);
568                                 strcat(path, "=");
569                                 strcat(path, st_path);
570                                 strcat(path, "\n");
571                                 entries[num_lines] = path;
572 #endif /* ! codereview */
573                         } else {
574                                 entries[num_lines] = strdup(temp_line);
575                         }
576                         num_lines++;
577                 } else if (found_title && !found_orig_be) {
578                         tmp_entries[num_tmp_lines] = strdup(temp_line);
579                         num_tmp_lines++;
580                 }
581         }
582
583         (void) fclose(menu_fp);
584
585         if (found_be) {
586                 /*
587                  * If an entry for this BE was already in the menu, then if
588                  * that entry's title matches what we would have put in
589                  * return success.  Otherwise return failure.
590                  */
591                 char *new_title = description ? description : be_name;
592
593                 if (strcmp(title, new_title) == 0) {
594                         ret = BE_SUCCESS;
595                         goto cleanup;
596                 } else {
597                         if (be_remove_menu(be_name, be_root_pool,
```

```
598                             boot_pool) != BE_SUCCESS) {
599                                 be_print_err(gettext("be_append_menu: "
600                                     "Failed to remove existing unusable "
601                                     "entry '%s' in boot menu.\n"), be_name);
602                                 ret = BE_ERR_BE_EXISTS;
603                                 goto cleanup;
604                         }
605                 }
606         }

608         /* Append BE entry to the end of the file */
609         menu_fp = fopen(menu_file, "a+");
610         err = errno;
611         if (menu_fp == NULL) {
612                 be_print_err(gettext("be_append_menu: failed "
613                     "to open menu.lst file %s\n"), menu_file);
614                 ret = errno_to_be_err(err);
615                 goto cleanup;
616         }

618         if (found_orig_be) {
619                 /*
620                  * write out all the stored lines
621                  */
622                 for (i = 0; i < num_lines; i++) {
623                         (void) fprintf(menu_fp, "%s", entries[i]);
624                         free(entries[i]);
625                 }
626                 num_lines = 0;

628                 /*
629                  * Check to see if this system supports grub
630                  */
631                 if (be_has_grub())
632                         (void) fprintf(menu_fp, "%s\n", BE_GRUB_COMMENT);
633                 ret = BE_SUCCESS;
634         } else {
635                 zprop_get_cbdata_t cb = { 0 };
636                 zprop_source_t src;
637                 char * bc = strchr(be_root_ds, '/');
638                 char sguid[] = "guid";

640                 (void) fprintf(menu_fp, "entry_name=%s\n",
639                 (void) fprintf(menu_fp, "title %s\n",
641                     description ? description : be_name);
642                 *bc = '\0';
643                 cb.cb_first = B_TRUE;
644                 cb.cb_sources = ZPROP_SRC_ALL;
645                 cb.cb_type = ZFS_TYPE_POOL;
646                 zprop_get_list(g_zfs, sguid, &cb.cb_proplist, ZFS_TYPE_POOL);
647                 zpool_handle_t * z_hndl = zpool_open(g_zfs, be_root_ds);
648                 *bc = '/';
649                 if (z_hndl) {
650                         uint64_t guid = zpool_get_prop_int(z_hndl, cb.cb_proplis
651                         (void) fprintf(menu_fp, "pool_uuid=%llx\n", guid);
652                         zpool_close(z_hndl);
653                 }
654                 (void) fprintf(menu_fp, "data_set=%s\n", be_root_ds);
541                 (void) fprintf(menu_fp, "bootfs %s\n", be_root_ds);

656                 /*
657                  * Check to see if this system supports grub
658                  */
659                 if (be_has_grub()) {
660                         (void) fprintf(menu_fp, "kernel_path="
661                             "/platform/i86pc/kernel/$ISADIR/unix\n");
```



```
662                         (void) fprintf(menu_fp, "kernel_options="
663                             "-B $ZFS_BOOTFS,console=graphic\n");
664                         (void) fprintf(menu_fp, "module="
547                         (void) fprintf(menu_fp, "kernel$ "
548                             "/platform/i86pc/kernel/$ISADIR/unix -B "
549                             "$ZFS-BOOTFS\n");
550                         (void) fprintf(menu_fp, "module$ "
665                             "/platform/i86pc/$ISADIR/boot_archive\n");
666                         (void) fprintf(menu_fp, "%s\n", BE_GRUB_COMMENT);
667                 }
668                 ret = BE_SUCCESS;
669         }
670         (void) fclose(menu_fp);
671 cleanup:
672         if (pool_mounted) {
673                 int err = BE_SUCCESS;
674                 err = be_unmount_pool(zhp, ptmp_mntpnt, orig_mntpnt);
675                 if (ret == BE_SUCCESS)
676                         ret = err;
677                 free(orig_mntpnt);
678                 free(ptmp_mntpnt);
679         }
680         ZFS_CLOSE(zhp);
681         if (num_tmp_lines > 0) {
682                 for (i = 0; i < num_tmp_lines; i++) {
683                         free(tmp_entries[i]);
684                         tmp_entries[i] = NULL;
685                 }
686         }
687         if (num_lines > 0) {
688                 for (i = 0; i < num_lines; i++) {
689                         free(entries[i]);
690                         entries[i] = NULL;
691                 }
692         }
693         return (ret);
694 }

696 /*
697  * Function:    be_remove_menu
698  * Description: Removes a BE's entry from a menu.lst file.
699  * Parameters:
700  *              be_name - the name of BE whose entry is to be removed from
701  *                      the menu.lst file.
702  *              be_root_pool - the pool that be_name lives in.
703  *              boot_pool - the pool where the BE is, if different than
704  *                      the pool containing the boot menu.  If this is
705  *                      NULL it will be set to be_root_pool.
706  * Returns:
707  *              BE_SUCCESS - Success
708  *              be_errno_t - Failure
709  * Scope:
710  *              Semi-private (library wide use only)
711  */
712 int
713 be_remove_menu(char *be_name, char *be_root_pool, char *boot_pool)
714 {
715         zfs_handle_t    *zhp = NULL;
716         char            be_root_ds[MAXPATHLEN];
717         char            **buffer = NULL;
718         char            menu_buf[BUFSIZ];
719         char            menu[MAXPATHLEN];
720         char            *pool_mntpnt = NULL;
721         char            *ptmp_mntpnt = NULL;
722         char            *orig_mntpnt = NULL;
723         char            *tmp_menu = NULL;
```

```
724         FILE            *menu_fp = NULL;
725         FILE            *tmp_menu_fp = NULL;
726         struct stat     sb;
727         int             ret = BE_SUCCESS;
728         int             i;
729         int             fd;
730         int             err = 0;
731         int             nlines = 0;
732         int             default_entry = 0;
733         int             entry_cnt = 0;
734         int             entry_del = 0;
735         int             num_entry_del = 0;
736         int             tmp_menu_len = 0;
737         boolean_t       write = B_TRUE;
738         boolean_t       do_buffer = B_FALSE;
739         boolean_t       pool_mounted = B_FALSE;

741         if (boot_pool == NULL)
742                 boot_pool = be_root_pool;

744         /* Get name of BE's root dataset */
745         be_make_root_ds(be_root_pool, be_name, be_root_ds, sizeof (be_root_ds));

747         /* Get handle to pool dataset */
748         if ((zhp = zfs_open(g_zfs, be_root_pool, ZFS_TYPE_DATASET)) == NULL) {
749                 be_print_err(gettext("be_remove_menu: "
750                     "failed to open pool dataset for %s: %s"),
751                     be_root_pool, libzfs_error_description(g_zfs));
752                 return (zfs_err_to_be_err(g_zfs));
753         }

755         /*
756          * Check to see if the pool's dataset is mounted. If it isn't we'll
757          * attempt to mount it.
758          */
759         if ((ret = be_mount_pool(zhp, &ptmp_mntpnt, &orig_mntpnt,
760             &pool_mounted)) != BE_SUCCESS) {
761                 be_print_err(gettext("be_remove_menu: pool dataset "
762                     "(%s) could not be mounted\n"), be_root_pool);
763                 ZFS_CLOSE(zhp);
764                 return (ret);
765         }

767         /*
768          * Get the mountpoint for the root pool dataset.
769          */
770         if (!zfs_is_mounted(zhp, &pool_mntpnt)) {
771                 be_print_err(gettext("be_remove_menu: pool "
772                     "dataset (%s) is not mounted. Can't set "
773                     "the default BE in the grub menu.\n"), be_root_pool);
774                 ret = BE_ERR_NO_MENU;
775                 goto cleanup;
776         }

778         /* Get path to boot menu */
779         (void) strlcpy(menu, pool_mntpnt, sizeof (menu));

781         /*
782          * Check to see if this system supports grub
783          */
784         if (be_has_grub())
785                 (void) strlcat(menu, BE_GRUB_MENU, sizeof (menu));
786         else
787                 (void) strlcat(menu, BE_SPARC_MENU, sizeof (menu));

789         /* Get handle to boot menu file */
```



```
790         if ((ret = be_open_menu(be_root_pool, menu, &menu_fp, "r",
791             B_TRUE)) != BE_SUCCESS) {
792                 goto cleanup;
793         } else if (menu_fp == NULL) {
794                 ret = BE_ERR_NO_MENU;
795                 goto cleanup;
796         }

798         free(pool_mntpnt);
799         pool_mntpnt = NULL;

801         /* Grab the stats of the original menu file */
802         if (stat(menu, &sb) != 0) {
803                 err = errno;
804                 be_print_err(gettext("be_remove_menu: "
805                     "failed to stat file %s: %s\n"), menu, strerror(err));
806                 ret = errno_to_be_err(err);
807                 goto cleanup;
808         }

810         /* Create a tmp file for the modified menu.lst */
811         tmp_menu_len = strlen(menu) + 7;
812         if ((tmp_menu = (char *)malloc(tmp_menu_len)) == NULL) {
813                 be_print_err(gettext("be_remove_menu: malloc failed\n"));
814                 ret = BE_ERR_NOMEM;
815                 goto cleanup;
816         }
817         (void) memset(tmp_menu, 0, tmp_menu_len);
818         (void) strlcpy(tmp_menu, menu, tmp_menu_len);
819         (void) strlcat(tmp_menu, "XXXXXX", tmp_menu_len);
820         if ((fd = mkstemp(tmp_menu)) == -1) {
821                 err = errno;
822                 be_print_err(gettext("be_remove_menu: mkstemp failed\n"));
823                 ret = errno_to_be_err(err);
824                 free(tmp_menu);
825                 tmp_menu = NULL;
826                 goto cleanup;
827         }
828         if ((tmp_menu_fp = fdopen(fd, "w")) == NULL) {
829                 err = errno;
830                 be_print_err(gettext("be_remove_menu: "
831                     "could not open tmp file for write: %s\n"), strerror(err));
832                 (void) close(fd);
833                 ret = errno_to_be_err(err);
834                 goto cleanup;
835         }

837         while (fgets(menu_buf, BUFSIZ, menu_fp)) {
838                 char tline [BUFSIZ];
839                 char *tok = NULL;

841                 (void) strlcpy(tline, menu_buf, sizeof (tline));

843                 /* Tokenize line */
844                 tok = strtok(tline, "=\n");
730                 tok = strtok(tline, BE_WHITE_SPACE);

846                 if (tok == NULL || tok[0] == '#') {
847                         /* Found empty line or comment line */
848                         if (do_buffer) {
849                                 /* Buffer this line */
850                                 if ((buffer = (char **)realloc(buffer,
851                                     sizeof (char *)*(nlines + 1))) == NULL) {
852                                         ret = BE_ERR_NOMEM;
853                                         goto cleanup;
854                                 }
```

```
855                             if ((buffer[nlines++] = strdup(menu_buf))
856                                 == NULL) {
857                                     ret = BE_ERR_NOMEM;
858                                     goto cleanup;
859                             }

861                     } else if (write || strncmp(menu_buf, BE_GRUB_COMMENT,
862                         strlen(BE_GRUB_COMMENT)) != 0) {
863                             /* Write this line out */
864                             (void) fputs(menu_buf, tmp_menu_fp);
865                     }
866             } else if (strcmp(tok, "default_entry") == 0) {
752             } else if (strcmp(tok, "default") == 0) {
867                     /*
868                      * Record what 'default' is set to because we might
869                      * need to adjust this upon deleting an entry.
870                      */
871                     tok = strtok(NULL, BE_WHITE_SPACE);

873                     if (tok != NULL) {
874                             default_entry = atoi(tok);
875                     }

877                     (void) fputs(menu_buf, tmp_menu_fp);
878             } else if (strcmp(tok, "entry_name") == 0) {
764             } else if (strcmp(tok, "title") == 0) {
879                     /*
880                      * If we've reached a 'title' line and do_buffer is
881                      * is true, that means we've just buffered an entire
882                      * entry without finding a 'bootfs' directive.  We
883                      * need to write that entry out and keep searching.
884                      */
885                     if (do_buffer) {
886                             for (i = 0; i < nlines; i++) {
887                                     (void) fputs(buffer[i], tmp_menu_fp);
888                                     free(buffer[i]);
889                             }
890                             free(buffer);
891                             buffer = NULL;
892                             nlines = 0;
893                     }

895                     /*
896                      * Turn writing off and buffering on, and increment
897                      * our entry counter.
898                      */
899                     write = B_FALSE;
900                     do_buffer = B_TRUE;
901                     entry_cnt++;

903                     /* Buffer this 'title' line */
904                     if ((buffer = (char **)realloc(buffer,
905                         sizeof (char *)*(nlines + 1))) == NULL) {
906                             ret = BE_ERR_NOMEM;
907                             goto cleanup;
908                     }
909                     if ((buffer[nlines++] = strdup(menu_buf)) == NULL) {
910                             ret = BE_ERR_NOMEM;
911                             goto cleanup;
912                     }

914             } else if (strcmp(tok, "data_set") == 0) {
800             } else if (strcmp(tok, "bootfs") == 0) {
915                     char *bootfs = NULL;

917                     /*
```



```
918                      * Found a 'bootfs' line.  See if it matches the
919                      * BE we're looking for.
920                      */
921                     if ((bootfs = strtok(NULL, BE_WHITE_SPACE)) == NULL ||
922                         strcmp(bootfs, be_root_ds) != 0) {
923                             /*
924                              * Either there's nothing after the 'bootfs'
925                              * or this is not the BE we're looking for,
926                              * write out the line(s) we've buffered since
927                              * finding the title.
928                              */
929                             for (i = 0; i < nlines; i++) {
930                                     (void) fputs(buffer[i], tmp_menu_fp);
931                                     free(buffer[i]);
932                             }
933                             free(buffer);
934                             buffer = NULL;
935                             nlines = 0;

937                             /*
938                              * Turn writing back on, and turn off buffering
939                              * since this isn't the entry we're looking
940                              * for.
941                              */
942                             write = B_TRUE;
943                             do_buffer = B_FALSE;

945                             /* Write this 'bootfs' line out. */
946                             (void) fputs(menu_buf, tmp_menu_fp);
947                     } else {
948                             /*
949                              * Found the entry we're looking for.
950                              * Record its entry number, increment the
951                              * number of entries we've deleted, and turn
952                              * writing off.  Also, throw away the lines
953                              * we've buffered for this entry so far, we
954                              * don't need them.
955                              */
956                             entry_del = entry_cnt - 1;
957                             num_entry_del++;
958                             write = B_FALSE;
959                             do_buffer = B_FALSE;

961                             for (i = 0; i < nlines; i++) {
962                                     free(buffer[i]);
963                             }
964                             free(buffer);
965                             buffer = NULL;
966                             nlines = 0;
967                     }
968             } else {
969                     if (do_buffer) {
970                             /* Buffer this line */
971                             if ((buffer = (char **)realloc(buffer,
972                                 sizeof (char *)*(nlines + 1))) == NULL) {
973                                     ret = BE_ERR_NOMEM;
974                                     goto cleanup;
975                             }
976                             if ((buffer[nlines++] = strdup(menu_buf))
977                                 == NULL) {
978                                     ret = BE_ERR_NOMEM;
979                                     goto cleanup;
980                             }
981                     } else if (write) {
982                             /* Write this line out */
983                             (void) fputs(menu_buf, tmp_menu_fp);
```

```
 984                         }
 985                 }
 986         }

 988         (void) fclose(menu_fp);
 989         menu_fp = NULL;
 990         (void) fclose(tmp_menu_fp);
 991         tmp_menu_fp = NULL;

 993         /* Copy the modified menu.lst into place */
 994         if (rename(tmp_menu, menu) != 0) {
 995                 err = errno;
 996                 be_print_err(gettext("be_remove_menu: "
 997                     "failed to rename file %s to %s: %s\n"),
 998                     tmp_menu, menu, strerror(err));
 999                 ret = errno_to_be_err(err);
1000                 goto cleanup;
1001         }
1002         free(tmp_menu);
1003         tmp_menu = NULL;

1005         /*
1006          * If we've removed an entry, see if we need to
1007          * adjust the default value in the menu.lst.  If the
1008          * entry we've deleted comes before the default entry
1009          * we need to adjust the default value accordingly.
1010          *
1011          * be_has_grub is used here to check to see if this system
1012          * supports grub.
1013          */
1014         if (be_has_grub() && num_entry_del > 0) {
1015                 if (entry_del <= default_entry) {
1016                         default_entry = default_entry - num_entry_del;
1017                         if (default_entry < 0)
1018                                 default_entry = 0;

1020                         /*
1021                          * Adjust the default value by rewriting the
1022                          * menu.lst file.  This may be overkill, but to
1023                          * preserve the location of the 'default' entry
1024                          * in the file, we need to do this.
1025                          */

1027                         /* Get handle to boot menu file */
1028                         if ((menu_fp = fopen(menu, "r")) == NULL) {
1029                                 err = errno;
1030                                 be_print_err(gettext("be_remove_menu: "
1031                                     "failed to open menu.lst (%s): %s\n"),
1032                                     menu, strerror(err));
1033                                 ret = errno_to_be_err(err);
1034                                 goto cleanup;
1035                         }

1037                         /* Create a tmp file for the modified menu.lst */
1038                         tmp_menu_len = strlen(menu) + 7;
1039                         if ((tmp_menu = (char *)malloc(tmp_menu_len))
1040                             == NULL) {
1041                                 be_print_err(gettext("be_remove_menu: "
1042                                     "malloc failed\n"));
1043                                 ret = BE_ERR_NOMEM;
1044                                 goto cleanup;
1045                         }
1046                         (void) memset(tmp_menu, 0, tmp_menu_len);
1047                         (void) strlcpy(tmp_menu, menu, tmp_menu_len);
1048                         (void) strlcat(tmp_menu, "XXXXXX", tmp_menu_len);
1049                         if ((fd = mkstemp(tmp_menu)) == -1) {
```



```
1050                                 err = errno;
1051                                 be_print_err(gettext("be_remove_menu: "
1052                                     "mkstemp failed: %s\n"), strerror(err));
1053                                 ret = errno_to_be_err(err);
1054                                 free(tmp_menu);
1055                                 tmp_menu = NULL;
1056                                 goto cleanup;
1057                         }
1058                         if ((tmp_menu_fp = fdopen(fd, "w")) == NULL) {
1059                                 err = errno;
1060                                 be_print_err(gettext("be_remove_menu: "
1061                                     "could not open tmp file for write: %s\n"),
1062                                     strerror(err));
1063                                 (void) close(fd);
1064                                 ret = errno_to_be_err(err);
1065                                 goto cleanup;
1066                         }

1068                         while (fgets(menu_buf, BUFSIZ, menu_fp)) {
1069                                 char tline [BUFSIZ];
1070                                 char *tok = NULL;

1072                                 (void) strlcpy(tline, menu_buf, sizeof (tline));

1074                                 /* Tokenize line */
1075                                 tok = strtok(tline, "=\n");
 961                                 tok = strtok(tline, BE_WHITE_SPACE);

1077                                 if (tok == NULL) {
1078                                         /* Found empty line, write it out */
1079                                         (void) fputs(menu_buf, tmp_menu_fp);
1080                                 } else if (strcmp(tok, "default_entry") == 0) {
 966                                 } else if (strcmp(tok, "default") == 0) {
1081                                         /* Found the default line, adjust it */
1082                                         (void) snprintf(tline, sizeof (tline),
1083                                             "default_entry=%d\n", default_entry)
 969                                             "default %d\n", default_entry);

1085                                         (void) fputs(tline, tmp_menu_fp);
1086                                 } else {
1087                                         /* Pass through all other lines */
1088                                         (void) fputs(menu_buf, tmp_menu_fp);
1089                                 }
1090                         }

1092                         (void) fclose(menu_fp);
1093                         menu_fp = NULL;
1094                         (void) fclose(tmp_menu_fp);
1095                         tmp_menu_fp = NULL;

1097                         /* Copy the modified menu.lst into place */
1098                         if (rename(tmp_menu, menu) != 0) {
1099                                 err = errno;
1100                                 be_print_err(gettext("be_remove_menu: "
1101                                     "failed to rename file %s to %s: %s\n"),
1102                                     tmp_menu, menu, strerror(err));
1103                                 ret = errno_to_be_err(err);
1104                                 goto cleanup;
1105                         }

1107                         free(tmp_menu);
1108                         tmp_menu = NULL;
1109                 }
1110         }

1112         /* Set the perms and ownership of the updated file */
```

```
1113         if (chmod(menu, sb.st_mode) != 0) {
1114                 err = errno;
1115                 be_print_err(gettext("be_remove_menu: "
1116                     "failed to chmod %s: %s\n"), menu, strerror(err));
1117                 ret = errno_to_be_err(err);
1118                 goto cleanup;
1119         }
1120         if (chown(menu, sb.st_uid, sb.st_gid) != 0) {
1121                 err = errno;
1122                 be_print_err(gettext("be_remove_menu: "
1123                     "failed to chown %s: %s\n"), menu, strerror(err));
1124                 ret = errno_to_be_err(err);
1125                 goto cleanup;
1126         }

1128 cleanup:
1129         if (pool_mounted) {
1130                 int err = BE_SUCCESS;
1131                 err = be_unmount_pool(zhp, ptmp_mntpnt, orig_mntpnt);
1132                 if (ret == BE_SUCCESS)
1133                         ret = err;
1134                 free(orig_mntpnt);
1135                 free(ptmp_mntpnt);
1136         }
1137         ZFS_CLOSE(zhp);

1139         free(buffer);
1140         if (menu_fp != NULL)
1141                 (void) fclose(menu_fp);
1142         if (tmp_menu_fp != NULL)
1143                 (void) fclose(tmp_menu_fp);
1144         if (tmp_menu != NULL) {
1145                 (void) unlink(tmp_menu);
1146                 free(tmp_menu);
1147         }

1149         return (ret);
1150 }

1152 /*
1153  * Function:    be_default_grub_bootfs
1154  * Description: This function returns the dataset in the default entry of
1155  *              the grub menu. If no default entry is found with a valid bootfs
1156  *              entry NULL is returned.
1157  * Parameters:
1158  *              be_root_pool - This is the name of the root pool where the
1159  *                             grub menu can be found.
1160  *              def_bootfs - This is used to pass back the bootfs string. On
1161  *                               error NULL is returned here.
1162  * Returns:
1163  *              Success - BE_SUCCESS is returned.
1164  *              Failure - a be_errno_t is returned.
1165  * Scope:
1166  *              Semi-private (library wide use only)
1167  */
1168 int
1169 be_default_grub_bootfs(const char *be_root_pool, char **def_bootfs)
1170 {
1171         zfs_handle_t    *zhp = NULL;
1172         char            grub_file[MAXPATHLEN];
1173         FILE            *menu_fp;
1174         char            line[BUFSIZ];
1175         char            *pool_mntpnt = NULL;
1176         char            *ptmp_mntpnt = NULL;
1177         char            *orig_mntpnt = NULL;
1178         int             default_entry = 0, entries = 0;
```



```
1179         int             found_default = 0;
1180         int             ret = BE_SUCCESS;
1181         boolean_t       pool_mounted = B_FALSE;

1183         errno = 0;

1185         /*
1186          * Check to see if this system supports grub
1187          */
1188         if (!be_has_grub()) {
1189                 be_print_err(gettext("be_default_grub_bootfs: operation "
1190                     "not supported on this architecture\n"));
1191                 return (BE_ERR_NOTSUP);
1192         }

1194         *def_bootfs = NULL;

1196         /* Get handle to pool dataset */
1197         if ((zhp = zfs_open(g_zfs, be_root_pool, ZFS_TYPE_DATASET)) == NULL) {
1198                 be_print_err(gettext("be_default_grub_bootfs: "
1199                     "failed to open pool dataset for %s: %s"),
1200                     be_root_pool, libzfs_error_description(g_zfs));
1201                 return (zfs_err_to_be_err(g_zfs));
1202         }

1204         /*
1205          * Check to see if the pool's dataset is mounted. If it isn't we'll
1206          * attempt to mount it.
1207          */
1208         if ((ret = be_mount_pool(zhp, &ptmp_mntpnt, &orig_mntpnt,
1209             &pool_mounted)) != BE_SUCCESS) {
1210                 be_print_err(gettext("be_default_grub_bootfs: pool dataset "
1211                     "(%s) could not be mounted\n"), be_root_pool);
1212                 ZFS_CLOSE(zhp);
1213                 return (ret);
1214         }

1216         /*
1217          * Get the mountpoint for the root pool dataset.
1218          */
1219         if (!zfs_is_mounted(zhp, &pool_mntpnt)) {
1220                 be_print_err(gettext("be_default_grub_bootfs: failed "
1221                     "to get mount point for the root pool. Can't set "
1222                     "the default BE in the grub menu.\n"));
1223                 ret = BE_ERR_NO_MENU;
1224                 goto cleanup;
1225         }

1227         (void) snprintf(grub_file, MAXPATHLEN, "%s%s",
1228             pool_mntpnt, BE_GRUB_MENU);

1230         if ((ret = be_open_menu((char *)be_root_pool, grub_file,
1231             &menu_fp, "r", B_FALSE)) != BE_SUCCESS) {
1232                 goto cleanup;
1233         } else if (menu_fp == NULL) {
1234                 ret = BE_ERR_NO_MENU;
1235                 goto cleanup;
1236         }

1238         free(pool_mntpnt);
1239         pool_mntpnt = NULL;

1241         while (fgets(line, BUFSIZ, menu_fp)) {
1242                 char *tok = strtok(line, "=\n");
1128                 char *tok = strtok(line, BE_WHITE_SPACE);
```

```
1244                 if (tok != NULL && tok[0] != '#') {
1245                         if (!found_default) {
1246                                 if (strcmp(tok, "default_entry") == 0) {
1132                                 if (strcmp(tok, "default") == 0) {
1247                                         tok = strtok(NULL, BE_WHITE_SPACE);
1248                                         if (tok != NULL) {
1249                                                 default_entry = atoi(tok);
1250                                                 rewind(menu_fp);
1251                                                 found_default = 1;
1252                                         }
1253                                 }
1254                                 continue;
1255                         }
1256                         if (strcmp(tok, "entry_name") == 0) {
1142                         if (strcmp(tok, "title") == 0) {
1257                                 entries++;
1258                         } else if (default_entry == entries - 1) {
1259                                 if (strcmp(tok, "data_set") == 0) {
1145                                 if (strcmp(tok, "bootfs") == 0) {
1260                                         tok = strtok(NULL, BE_WHITE_SPACE);
1261                                         (void) fclose(menu_fp);

1263                                         if (tok == NULL) {
1264                                                 ret = BE_SUCCESS;
1265                                                 goto cleanup;
1266                                         }

1268                                         if ((*def_bootfs = strdup(tok)) !=
1269                                             NULL) {
1270                                                 ret = BE_SUCCESS;
1271                                                 goto cleanup;
1272                                         }
1273                                         be_print_err(gettext(
1274                                             "be_default_grub_bootfs: "
1275                                             "memory allocation failed\n"));
1276                                         ret = BE_ERR_NOMEM;
1277                                         goto cleanup;
1278                                 }
1279                         } else if (default_entry < entries - 1) {
1280                                 /*
1281                                  * no bootfs entry for the default entry.
1282                                  */
1283                                 break;
1284                         }
1285                 }
1286         }
1287         (void) fclose(menu_fp);

1289 cleanup:
1290         if (pool_mounted) {
1291                 int err = BE_SUCCESS;
1292                 err = be_unmount_pool(zhp, ptmp_mntpnt, orig_mntpnt);
1293                 if (ret == BE_SUCCESS)
1294                         ret = err;
1295                 free(orig_mntpnt);
1296                 free(ptmp_mntpnt);
1297         }
1298         ZFS_CLOSE(zhp);
1299         return (ret);
1300 }

1302 /*
1303  * Function:    be_change_grub_default
1304  * Description: This function takes two parameters. These are the name of
1305  *              the BE we want to have as the default booted in the grub
1306  *              menu and the root pool where the path to the grub menu exists.
```



```
1307  *              The code takes this and finds the BE's entry in the grub menu
1308  *              and changes the default entry to point to that entry in the
1309  *              list.
1310  * Parameters:
1311  *              be_name - This is the name of the BE wanted as the default
1312  *                      for the next boot.
1313  *              be_root_pool - This is the name of the root pool where the
1314  *                      grub menu can be found.
1315  * Returns:
1316  *              BE_SUCCESS - Success
1317  *              be_errno_t - Failure
1318  * Scope:
1319  *              Semi-private (library wide use only)
1320  */
1321 int
1322 be_change_grub_default(char *be_name, char *be_root_pool)
1323 {
1324         zfs_handle_t    *zhp = NULL;
1325         char    grub_file[MAXPATHLEN];
1326         char    *temp_grub;
1327         char    *pool_mntpnt = NULL;
1328         char    *ptmp_mntpnt = NULL;
1329         char    *orig_mntpnt = NULL;
1330         char    line[BUFSIZ];
1331         char    temp_line[BUFSIZ];
1332         char    be_root_ds[MAXPATHLEN];
1333         FILE    *grub_fp = NULL;
1334         FILE    *temp_fp = NULL;
1335         struct stat     sb;
1336         int     temp_grub_len = 0;
1337         int     fd, entries = 0;
1338         int     err = 0;
1339         int     ret = BE_SUCCESS;
1340         boolean_t       found_default = B_FALSE;
1341         boolean_t       pool_mounted = B_FALSE;

1343         errno = 0;

1345         /*
1346          * Check to see if this system supports grub
1347          */
1348         if (!be_has_grub()) {
1349                 be_print_err(gettext("be_change_grub_default: operation "
1350                     "not supported on this architecture\n"));
1351                 return (BE_ERR_NOTSUP);
1352         }

1354         /* Generate string for BE's root dataset */
1355         be_make_root_ds(be_root_pool, be_name, be_root_ds, sizeof (be_root_ds));

1357         /* Get handle to pool dataset */
1358         if ((zhp = zfs_open(g_zfs, be_root_pool, ZFS_TYPE_DATASET)) == NULL) {
1359                 be_print_err(gettext("be_change_grub_default: "
1360                     "failed to open pool dataset for %s: %s"),
1361                     be_root_pool, libzfs_error_description(g_zfs));
1362                 return (zfs_err_to_be_err(g_zfs));
1363         }

1365         /*
1366          * Check to see if the pool's dataset is mounted. If it isn't we'll
1367          * attempt to mount it.
1368          */
1369         if ((ret = be_mount_pool(zhp, &ptmp_mntpnt, &orig_mntpnt,
1370             &pool_mounted)) != BE_SUCCESS) {
1371                 be_print_err(gettext("be_change_grub_default: pool dataset "
1372                     "(%s) could not be mounted\n"), be_root_pool);
```

```
1373                 ZFS_CLOSE(zhp);
1374                 return (ret);
1375         }

1377         /*
1378          * Get the mountpoint for the root pool dataset.
1379          */
1380         if (!zfs_is_mounted(zhp, &pool_mntpnt)) {
1381                 be_print_err(gettext("be_change_grub_default: pool "
1382                     "dataset (%s) is not mounted. Can't set "
1383                     "the default BE in the grub menu.\n"), be_root_pool);
1384                 ret = BE_ERR_NO_MENU;
1385                 goto cleanup;
1386         }

1388         (void) snprintf(grub_file, MAXPATHLEN, "%s%s",
1389             pool_mntpnt, BE_GRUB_MENU);

1391         if ((ret = be_open_menu(be_root_pool, grub_file,
1392             &grub_fp, "r+", B_TRUE)) != BE_SUCCESS) {
1393                 goto cleanup;
1394         } else if (grub_fp == NULL) {
1395                 ret = BE_ERR_NO_MENU;
1396                 goto cleanup;
1397         }

1399         free(pool_mntpnt);
1400         pool_mntpnt = NULL;

1402         /* Grab the stats of the original menu file */
1403         if (stat(grub_file, &sb) != 0) {
1404                 err = errno;
1405                 be_print_err(gettext("be_change_grub_default: "
1406                     "failed to stat file %s: %s\n"), grub_file, strerror(err));
1407                 ret = errno_to_be_err(err);
1408                 goto cleanup;
1409         }

1411         /* Create a tmp file for the modified menu.lst */
1412         temp_grub_len = strlen(grub_file) + 7;
1413         if ((temp_grub = (char *)malloc(temp_grub_len)) == NULL) {
1414                 be_print_err(gettext("be_change_grub_default: "
1415                     "malloc failed\n"));
1416                 ret = BE_ERR_NOMEM;
1417                 goto cleanup;
1418         }
1419         (void) memset(temp_grub, 0, temp_grub_len);
1420         (void) strlcpy(temp_grub, grub_file, temp_grub_len);
1421         (void) strlcat(temp_grub, "XXXXXX", temp_grub_len);
1422         if ((fd = mkstemp(temp_grub)) == -1) {
1423                 err = errno;
1424                 be_print_err(gettext("be_change_grub_default: "
1425                     "mkstemp failed: %s\n"), strerror(err));
1426                 ret = errno_to_be_err(err);
1427                 free(temp_grub);
1428                 temp_grub = NULL;
1429                 goto cleanup;
1430         }
1431         if ((temp_fp = fdopen(fd, "w")) == NULL) {
1432                 err = errno;
1433                 be_print_err(gettext("be_change_grub_default: "
1434                     "failed to open %s file: %s\n"),
1435                     temp_grub, strerror(err));
1436                 (void) close(fd);
1437                 ret = errno_to_be_err(err);
1438                 goto cleanup;
```



```
1439         }

1441         while (fgets(line, BUFSIZ, grub_fp)) {
1442                 char *tok = strtok(line, "=\n");
1328                 char *tok = strtok(line, BE_WHITE_SPACE);

1444                 if (tok == NULL || tok[0] == '#') {
1445                         continue;
1446                 } else if (strcmp(tok, "entry_name") == 0) {
1332                 } else if (strcmp(tok, "title") == 0) {
1447                         entries++;
1448                         continue;
1449                 } else if (strcmp(tok, "data_set") == 0) {
1335                 } else if (strcmp(tok, "bootfs") == 0) {
1450                         char *bootfs = strtok(NULL, BE_WHITE_SPACE);
1451                         if (bootfs == NULL)
1452                                 continue;

1454                         if (strcmp(bootfs, be_root_ds) == 0) {
1455                                 found_default = B_TRUE;
1456                                 break;
1457                         }
1458                 }
1459         }

1461         if (!found_default) {
1462                 be_print_err(gettext("be_change_grub_default: failed "
1463                     "to find entry for %s in the grub menu\n"),
1464                     be_name);
1465                 ret = BE_ERR_BE_NOENT;
1466                 goto cleanup;
1467         }

1469         rewind(grub_fp);

1471         (void) snprintf(temp_line, BUFSIZ, "default_entry=%d\n",
1472             entries - 1 >= 0 ? entries - 1 : 0);
1473         (void) fputs(temp_line, temp_fp);
1474 #endif /* ! codereview */
1475         while (fgets(line, BUFSIZ, grub_fp)) {
1476                 char *tok = NULL;

1478                 (void) strncpy(temp_line, line, BUFSIZ);

1480                 if ((tok = strtok(temp_line, "=\n")) != NULL &&
1481                     strcmp(tok, "default_entry") == 0) {
1357                 if ((tok = strtok(temp_line, BE_WHITE_SPACE)) != NULL &&
1358                     strcmp(tok, "default") == 0) {
1359                         (void) snprintf(temp_line, BUFSIZ, "default %d\n",
1360                             entries - 1 >= 0 ? entries - 1 : 0);
1361                         (void) fputs(temp_line, temp_fp);
1482                 } else {
1483                         (void) fputs(line, temp_fp);
1484                 }
1485         }

1487         (void) fclose(grub_fp);
1488         grub_fp = NULL;
1489         (void) fclose(temp_fp);
1490         temp_fp = NULL;

1492         if (rename(temp_grub, grub_file) != 0) {
1493                 err = errno;
1494                 be_print_err(gettext("be_change_grub_default: "
1495                     "failed to rename file %s to %s: %s\n"),
1496                     temp_grub, grub_file, strerror(err));
```

```
1497                 ret = errno_to_be_err(err);
1498                 goto cleanup;
1499         }
1500         free(temp_grub);
1501         temp_grub = NULL;

1503         /* Set the perms and ownership of the updated file */
1504         if (chmod(grub_file, sb.st_mode) != 0) {
1505                 err = errno;
1506                 be_print_err(gettext("be_change_grub_default: "
1507                     "failed to chmod %s: %s\n"), grub_file, strerror(err));
1508                 ret = errno_to_be_err(err);
1509                 goto cleanup;
1510         }
1511         if (chown(grub_file, sb.st_uid, sb.st_gid) != 0) {
1512                 err = errno;
1513                 be_print_err(gettext("be_change_grub_default: "
1514                     "failed to chown %s: %s\n"), grub_file, strerror(err));
1515                 ret = errno_to_be_err(err);
1516         }

1518 cleanup:
1519         if (pool_mounted) {
1520                 int err = BE_SUCCESS;
1521                 err = be_unmount_pool(zhp, ptmp_mntpnt, orig_mntpnt);
1522                 if (ret == BE_SUCCESS)
1523                         ret = err;
1524                 free(orig_mntpnt);
1525                 free(ptmp_mntpnt);
1526         }
1527         ZFS_CLOSE(zhp);
1528         if (grub_fp != NULL)
1529                 (void) fclose(grub_fp);
1530         if (temp_fp != NULL)
1531                 (void) fclose(temp_fp);
1532         if (temp_grub != NULL) {
1533                 (void) unlink(temp_grub);
1534                 free(temp_grub);
1535         }

1537         return (ret);
1538 }

1540 /*
1541  * Function:    be_update_menu
1542  * Description: This function is used by be_rename to change the BE name in
1543  *              an existing entry in the grub menu to the new name of the BE.
1544  * Parameters:
1545  *              be_orig_name - the original name of the BE
1546  *              be_new_name - the new name the BE is being renameed to.
1547  *              be_root_pool - The pool which contains the grub menu
1548  *              boot_pool - the pool where the BE is, if different than
1549  *                      the pool containing the boot menu.  If this is
1550  *                      NULL it will be set to be_root_pool.
1551  * Returns:
1552  *              BE_SUCCESS - Success
1553  *              be_errno_t - Failure
1554  * Scope:
1555  *              Semi-private (library wide use only)
1556  */
1557 int
1558 be_update_menu(char *be_orig_name, char *be_new_name, char *be_root_pool,
1559     char *boot_pool)
1560 {
1561         zfs_handle_t *zhp = NULL;
1562         char menu_file[MAXPATHLEN];
```



```
1563         char be_root_ds[MAXPATHLEN];
1564         char be_new_root_ds[MAXPATHLEN];
1565         char line[BUFSIZ];
1566         char *pool_mntpnt = NULL;
1567         char *ptmp_mntpnt = NULL;
1568         char *orig_mntpnt = NULL;
1569         char *temp_menu = NULL;
1570         FILE *menu_fp = NULL;
1571         FILE *new_fp = NULL;
1572         struct stat sb;
1573         int temp_menu_len = 0;
1574         int tmp_fd;
1575         int ret = BE_SUCCESS;
1576         int flag = 0;
1577 #endif /* ! codereview */
1578         int err = 0;
1579         boolean_t pool_mounted = B_FALSE;

1581         errno = 0;

1583         if (boot_pool == NULL)
1584                 boot_pool = be_root_pool;

1586         if ((zhp = zfs_open(g_zfs, be_root_pool, ZFS_TYPE_DATASET)) == NULL) {
1587                 be_print_err(gettext("be_update_menu: failed to open "
1588                     "pool dataset for %s: %s\n"), be_root_pool,
1589                     libzfs_error_description(g_zfs));
1590                 return (zfs_err_to_be_err(g_zfs));
1591         }

1593         /*
1594          * Check to see if the pool's dataset is mounted. If it isn't we'll
1595          * attempt to mount it.
1596          */
1597         if ((ret = be_mount_pool(zhp, &ptmp_mntpnt, &orig_mntpnt,
1598             &pool_mounted)) != BE_SUCCESS) {
1599                 be_print_err(gettext("be_update_menu: pool dataset "
1600                     "(%s) could not be mounted\n"), be_root_pool);
1601                 ZFS_CLOSE(zhp);
1602                 return (ret);
1603         }

1605         /*
1606          * Get the mountpoint for the root pool dataset.
1607          */
1608         if (!zfs_is_mounted(zhp, &pool_mntpnt)) {
1609                 be_print_err(gettext("be_update_menu: failed "
1610                     "to get mount point for the root pool. Can't set "
1611                     "the default BE in the grub menu.\n"));
1612                 ret = BE_ERR_NO_MENU;
1613                 goto cleanup;
1614         }

1616         /*
1617          * Check to see if this system supports grub
1618          */
1619         if (be_has_grub()) {
1620                 (void) snprintf(menu_file, sizeof (menu_file),
1621                     "%s%s", pool_mntpnt, BE_GRUB_MENU);
1622         } else {
1623                 (void) snprintf(menu_file, sizeof (menu_file),
1624                     "%s%s", pool_mntpnt, BE_SPARC_MENU);
1625         }

1627         be_make_root_ds(be_root_pool, be_orig_name, be_root_ds,
1628             sizeof (be_root_ds));
```

```
1629         be_make_root_ds(be_root_pool, be_new_name, be_new_root_ds,
1630             sizeof (be_new_root_ds));

1632         if ((ret = be_open_menu(be_root_pool, menu_file,
1633             &menu_fp, "r", B_TRUE)) != BE_SUCCESS) {
1634                 goto cleanup;
1635         } else if (menu_fp == NULL) {
1636                 ret = BE_ERR_NO_MENU;
1637                 goto cleanup;
1638         }

1640         free(pool_mntpnt);
1641         pool_mntpnt = NULL;

1643         /* Grab the stat of the original menu file */
1644         if (stat(menu_file, &sb) != 0) {
1645                 err = errno;
1646                 be_print_err(gettext("be_update_menu: "
1647                     "failed to stat file %s: %s\n"), menu_file, strerror(err));
1648                 (void) fclose(menu_fp);
1649                 ret = errno_to_be_err(err);
1650                 goto cleanup;
1651         }

1653         /* Create tmp file for modified menu.lst */
1654         temp_menu_len = strlen(menu_file) + 7;
1655         if ((temp_menu = (char *)malloc(temp_menu_len))
1656             == NULL) {
1657                 be_print_err(gettext("be_update_menu: "
1658                     "malloc failed\n"));
1659                 (void) fclose(menu_fp);
1660                 ret = BE_ERR_NOMEM;
1661                 goto cleanup;
1662         }
1663         (void) memset(temp_menu, 0, temp_menu_len);
1664         (void) strlcpy(temp_menu, menu_file, temp_menu_len);
1665         (void) strlcat(temp_menu, "XXXXXX", temp_menu_len);
1666         if ((tmp_fd = mkstemp(temp_menu)) == -1) {
1667                 err = errno;
1668                 be_print_err(gettext("be_update_menu: "
1669                     "mkstemp failed: %s\n"), strerror(err));
1670                 (void) fclose(menu_fp);
1671                 free(temp_menu);
1672                 ret = errno_to_be_err(err);
1673                 goto cleanup;
1674         }
1675         if ((new_fp = fdopen(tmp_fd, "w")) == NULL) {
1676                 err = errno;
1677                 be_print_err(gettext("be_update_menu: "
1678                     "fdopen failed: %s\n"), strerror(err));
1679                 (void) close(tmp_fd);
1680                 (void) fclose(menu_fp);
1681                 free(temp_menu);
1682                 ret = errno_to_be_err(err);
1683                 goto cleanup;
1684         }

1686         while (fgets(line, BUFSIZ, menu_fp)) {
1687                 char tline[BUFSIZ];
1688                 char new_line[BUFSIZ];
1689                 char *c = NULL;

1691                 (void) strlcpy(tline, line, sizeof (tline));

1693                 /* Tokenize line */
1694                 c = strtok(tline, "=\n");
```



```
1456                 c = strtok(tline, BE_WHITE_SPACE);

1696                 if (c == NULL) {
1697                         /* Found empty line, write it out. */
1698                         (void) fputs(line, new_fp);
1699                 } else if (c[0] == '#') {
1700                         /* Found a comment line, write it out. */
1701                         (void) fputs(line, new_fp);
1702                 } else if (strcmp(c, "entry_name") == 0) {
1464                 } else if (strcmp(c, "title") == 0) {
1703                         char *name = NULL;
1704                         char *desc = NULL;

1706                         /*
1707                          * Found a 'title' line, parse out BE name or
1708                          * the description.
1709                          */
1710                         flag = 0;
1711 #endif /* ! codereview */
1712                         name = strtok(NULL, BE_WHITE_SPACE);

1714                         if (name == NULL) {
1715                                 /*
1716                                  * Nothing after 'title', just push
1717                                  * this line through
1718                                  */
1719                                 (void) fputs(line, new_fp);
1720                         } else {
1721                                 /*
1722                                  * Grab the remainder of the title which
1723                                  * could be a multi worded description
1724                                  */
1725                                 desc = strtok(NULL, "\n");

1727                                 if (strcmp(name, be_orig_name) == 0) {
1728                                         /*
1729                                          * The first token of the title is
1730                                          * the old BE name, replace it with
1731                                          * the new one, and write it out
1732                                          * along with the remainder of
1733                                          * description if there is one.
1734                                          */
1735                                         ++flag;
1736 #endif /* ! codereview */
1737                                         if (desc) {
1738                                                 (void) snprintf(new_line,
1739                                                     sizeof (new_line),
1740                                                     "entry_name=%s %s\n",
1472                                                     "title %s %s\n",
1741                                                     be_new_name, desc);
1742                                         } else {
1743                                                 (void) snprintf(new_line,
1744                                                     sizeof (new_line),
1745                                                     "entry_name=%s\n", be_new_na
1477                                                     "title %s\n", be_new_name);
1746                                         }

1748                                         (void) fputs(new_line, new_fp);
1749                                 } else {
1750                                         (void) fputs(line, new_fp);
1751                                 }
1752                         }
1753                 } else if (strcmp(c, "data_set") == 0) {
1485                 } else if (strcmp(c, "bootfs") == 0) {
1754                         /*
1755                          * Found a 'bootfs' line, parse out the BE root
```

```
1756                          * dataset value.
1757                          */
1758                         char *root_ds = strtok(NULL, BE_WHITE_SPACE);

1760                         if (root_ds == NULL) {
1761                                 /*
1762                                  * Nothing after 'bootfs', just push
1763                                  * this line through
1764                                  */
1765                                 (void) fputs(line, new_fp);
1766                         } else {
1767                                 /*
1768                                  * If this bootfs is the one we're renaming,
1769                                  * write out the new root dataset value
1770                                  */
1771                                 if (strcmp(root_ds, be_root_ds) == 0) {
1772                                         ++flag;
1773 #endif /* ! codereview */
1774                                         (void) snprintf(new_line,
1775                                             sizeof (new_line), "data_set=%s\n",
1504                                             sizeof (new_line), "bootfs %s\n",
1776                                             be_new_root_ds);

1778                                         (void) fputs(new_line, new_fp);
1779                                 } else {
1780                                         (void) fputs(line, new_fp);
1781                                 }
1782                         }
1783                 } else {
1784                         /*
1785                          * Found some other line we don't care
1786                          * about, write it out.
1787                          */
1788                         (void) fputs(line, new_fp);
1789                 }
1790         }

1792         (void) fclose(menu_fp);
1793         (void) fclose(new_fp);
1794         (void) close(tmp_fd);

1796         if (rename(temp_menu, menu_file) != 0) {
1797                 err = errno;
1798                 be_print_err(gettext("be_update_menu: "
1799                     "failed to rename file %s to %s: %s\n"),
1800                     temp_menu, menu_file, strerror(err));
1801                 ret = errno_to_be_err(err);
1802         }
1803         free(temp_menu);

1805         /* Set the perms and ownership of the updated file */
1806         if (chmod(menu_file, sb.st_mode) != 0) {
1807                 err = errno;
1808                 be_print_err(gettext("be_update_menu: "
1809                     "failed to chmod %s: %s\n"), menu_file, strerror(err));
1810                 ret = errno_to_be_err(err);
1811                 goto cleanup;
1812         }
1813         if (chown(menu_file, sb.st_uid, sb.st_gid) != 0) {
1814                 err = errno;
1815                 be_print_err(gettext("be_update_menu: "
1816                     "failed to chown %s: %s\n"), menu_file, strerror(err));
1817                 ret = errno_to_be_err(err);
1818         }

1820 cleanup:
```



```
1821         if (pool_mounted) {
1822                 int err = BE_SUCCESS;
1823                 err = be_unmount_pool(zhp, ptmp_mntpnt, orig_mntpnt);
1824                 if (ret == BE_SUCCESS)
1825                         ret = err;
1826                 free(orig_mntpnt);
1827                 free(ptmp_mntpnt);
1828         }
1829         ZFS_CLOSE(zhp);
1830         return (ret);
1831 }

1833 /*
1834  * Function:    be_has_menu_entry
1835  * Description: Checks to see if the BEs root dataset has an entry in the grub
1836  *              menu.
1837  * Parameters:
1838  *              be_dataset - The root dataset of the BE
1839  *              be_root_pool - The pool which contains the boot menu
1840  *              entry - A pointer the the entry number of the BE if found.
1841  * Returns:
1842  *              B_TRUE - Success
1843  *              B_FALSE - Failure
1844  * Scope:
1845  *              Semi-private (library wide use only)
1846  */
1847 boolean_t
1848 be_has_menu_entry(char *be_dataset, char *be_root_pool, int *entry)
1849 {
1850         zfs_handle_t *zhp = NULL;
1851         char            menu_file[MAXPATHLEN];
1852         FILE            *menu_fp;
1853         char            line[BUFSIZ];
1854         char            *last;
1855         char            *rpool_mntpnt = NULL;
1856         char            *ptmp_mntpnt = NULL;
1857         char            *orig_mntpnt = NULL;
1858         int             ent_num = 0;
1859         boolean_t       ret = 0;
1860         boolean_t       pool_mounted = B_FALSE;


1863         /*
1864          * Check to see if this system supports grub
1865          */
1866         if ((zhp = zfs_open(g_zfs, be_root_pool, ZFS_TYPE_DATASET)) == NULL) {
1867                 be_print_err(gettext("be_has_menu_entry: failed to open "
1868                     "pool dataset for %s: %s\n"), be_root_pool,
1869                     libzfs_error_description(g_zfs));
1870                 return (B_FALSE);
1871         }

1873         /*
1874          * Check to see if the pool's dataset is mounted. If it isn't we'll
1875          * attempt to mount it.
1876          */
1877         if (be_mount_pool(zhp, &ptmp_mntpnt, &orig_mntpnt,
1878             &pool_mounted) != 0) {
1879                 be_print_err(gettext("be_has_menu_entry: pool dataset "
1880                     "(%s) could not be mounted\n"), be_root_pool);
1881                 ZFS_CLOSE(zhp);
1882                 return (B_FALSE);
1883         }

1885         /*
1886          * Get the mountpoint for the root pool dataset.
```

```
1887          */
1888         if (!zfs_is_mounted(zhp, &rpool_mntpnt)) {
1889                 be_print_err(gettext("be_has_menu_entry: pool "
1890                     "dataset (%s) is not mounted. Can't set "
1891                     "the default BE in the grub menu.\n"), be_root_pool);
1892                 ret = B_FALSE;
1893                 goto cleanup;
1894         }

1896         if (be_has_grub()) {
1897                 (void) snprintf(menu_file, MAXPATHLEN, "/%s%s",
1898                     rpool_mntpnt, BE_GRUB_MENU);
1899         } else {
1900                 (void) snprintf(menu_file, MAXPATHLEN, "/%s%s",
1901                     rpool_mntpnt, BE_SPARC_MENU);
1902         }

1904         if (be_open_menu(be_root_pool, menu_file, &menu_fp, "r",
1905             B_FALSE) != 0) {
1906                 ret = B_FALSE;
1907                 goto cleanup;
1908         } else if (menu_fp == NULL) {
1909                 ret = B_FALSE;
1910                 goto cleanup;
1911         }

1913         free(rpool_mntpnt);
1914         rpool_mntpnt = NULL;

1916         while (fgets(line, BUFSIZ, menu_fp)) {
1917                 char *tok = strtok_r(line, "=\n", &last);
1646                 char *tok = strtok_r(line, BE_WHITE_SPACE, &last);

1919                 if (tok != NULL && tok[0] != '#') {
1920                         if (strcmp(tok, "data_set") == 0) {
1649                         if (strcmp(tok, "bootfs") == 0) {
1921                                 tok = strtok_r(last, BE_WHITE_SPACE, &last);
1922                                 if (tok != NULL && strcmp(tok,
1923                                     be_dataset) == 0) {
1924                                         (void) fclose(menu_fp);
1925                                         /*
1926                                          * The entry number needs to be
1927                                          * decremented here because the title
1928                                          * will always be the first line for
1929                                          * an entry. Because of this we'll
1930                                          * always be off by one entry when we
1931                                          * check for bootfs.
1932                                          */
1933                                         *entry = ent_num - 1;
1934                                         ret = B_TRUE;
1935                                         goto cleanup;
1936                                 }
1937                         } else if (strcmp(tok, "entry_name") == 0)
1666                         } else if (strcmp(tok, "title") == 0)
1938                                 ent_num++;
1939                 }
1940         }

1942 cleanup:
1943         if (pool_mounted) {
1944                 (void) be_unmount_pool(zhp, ptmp_mntpnt, orig_mntpnt);
1945                 free(orig_mntpnt);
1946                 free(ptmp_mntpnt);
1947         }
1948         ZFS_CLOSE(zhp);
1949         (void) fclose(menu_fp);
```



```
1950         return (ret);
1951 }
_____unchanged_portion_omitted_

3624 /*
3625  * Function:    be_create_menu
3626  * Description:
3627  *              This function is used if no menu.lst file exists. In
3628  *              this case a new file is created and if needed default
3629  *              lines are added to the file.
3630  * Parameters:
3631  *              pool - The name of the pool the menu.lst file is on
3632  *              menu_file - The name of the file we're creating.
3633  *              menu_fp - A pointer to the file pointer of the file we
3634  *                        created. This is also used to pass back the file
3635  *                        pointer to the newly created file.
3636  *              mode - the original mode used for the failed attempt to
3637  *                     non-existent file.
3638  * Returns:
3639  *              BE_SUCCESS - Success
3640  *              be_errno_t - Failure
3641  * Scope:
3642  *              Private
3643  */
3644 static int
3645 be_create_menu(
3646         char *pool,
3647         char *menu_file,
3648         FILE **menu_fp,
3649         char *mode)
3650 {
3651         be_node_list_t  *be_nodes = NULL;
3652         char *menu_path = NULL;
3653         char *be_rpool = NULL;
3654         char *be_name = NULL;
3655         char *console = NULL;
3656         errno = 0;

3658         if (menu_file == NULL || menu_fp == NULL || mode == NULL)
3659                 return (BE_ERR_INVAL);

3661         menu_path = strdup(menu_file);
3662         if (menu_path == NULL)
3663                 return (BE_ERR_NOMEM);

3665         (void) dirname(menu_path);
3666         if (*menu_path == '.') {
3667                 free(menu_path);
3668                 return (BE_ERR_BAD_MENU_PATH);
3669         }
3670         if (mkdirp(menu_path,
3671             S_IRWXU | S_IRGRP | S_IXGRP | S_IROTH | S_IXOTH) == -1 &&
3672             errno != EEXIST) {
3673                 free(menu_path);
3674                 be_print_err(gettext("be_create_menu: Failed to create the %s "
3675                     "directory: %s\n"), menu_path, strerror(errno));
3676                 return (errno_to_be_err(errno));
3677         }
3678         free(menu_path);

3680         /*
3681          * Check to see if this system supports grub
3682          */
3683         if (be_has_grub()) {
3684                 /*
3685                  * The grub menu is missing so we need to create it
```

```
3686                 * and fill in the first few lines.
3687                 */
3688                FILE *temp_fp = fopen(menu_file, "a+");
3689                if (temp_fp == NULL) {
3690                        *menu_fp = NULL;
3691                        return (errno_to_be_err(errno));
3692                }

3694                if ((console = be_get_console_prop()) != NULL) {

3696                        /*
3697                         * If console is redirected to serial line,
3698                         * GRUB splash screen will not be enabled.
3699                         */
3700                        if (strncmp(console, "text", strlen("text")) == 0 ||
3701                            strncmp(console, "graphics",
3702                            strlen("graphics")) == 0) {
3703 /*

3704                                (void) fprintf(temp_fp, "%s\n", BE_GRUB_SPLASH);
3705                                (void) fprintf(temp_fp, "%s\n",
3706                                    BE_GRUB_FOREGROUND);
3707                                (void) fprintf(temp_fp, "%s\n",
3708                                    BE_GRUB_BACKGROUND);
3709                                (void) fprintf(temp_fp, "%s\n",
3710                                    BE_GRUB_DEFAULT);*/
3439                                    BE_GRUB_DEFAULT);
3711                        } else {
3712                                be_print_err(gettext("be_create_menu: "
3713                                    "console on serial line, "
3714                                    "GRUB splash image will be disabled\n"));
3715                        }
3716                }

3718                (void) fprintf(temp_fp, "timeout=30\n");
3447                (void) fprintf(temp_fp, "timeout 30\n");
3719                (void) fclose(temp_fp);

3721        } else {
3722                /*
3723                 * The menu file doesn't exist so we need to create a
3724                 * blank file.
3725                 */
3726                FILE *temp_fp = fopen(menu_file, "w+");
3727                if (temp_fp == NULL) {
3728                        *menu_fp = NULL;
3729                        return (errno_to_be_err(errno));
3730                }
3731                (void) fclose(temp_fp);
3732        }

3734        /*
3735         * Now we need to add all the BE's back into the the file.
3736         */
3737        if (_be_list(NULL, &be_nodes) == BE_SUCCESS) {
3738                while (be_nodes != NULL) {
3739                        if (strcmp(pool, be_nodes->be_rpool) == 0) {
3740                                (void) be_append_menu(be_nodes->be_node_name,
3741                                    be_nodes->be_rpool, NULL, NULL, NULL);
3742                        }
3743                        if (be_nodes->be_active_on_boot) {
3744                                be_rpool = strdup(be_nodes->be_rpool);
3745                                be_name = strdup(be_nodes->be_node_name);
3746                        }

3748                        be_nodes = be_nodes->be_next_node;
```



```
3749                }
3750        }
3751        be_free_list(be_nodes);

3753        /*
3754         * Check to see if this system supports grub
3755         */
3756        if (be_has_grub()) {
3757                int err = be_change_grub_default(be_name, be_rpool);
3758                if (err != BE_SUCCESS)
3759                        return (err);
3760        }
3761        *menu_fp = fopen(menu_file, mode);
3762        if (*menu_fp == NULL)
3763                return (errno_to_be_err(errno));

3765        return (BE_SUCCESS);
3766 }
```

*_____unchanged_portion_omitted_*