```
**********************************************************
  228220 Fri Jul 27 03:18:14 2012
new/usr/src/cmd/boot/bootadm/bootadm.c
backup vers
bootadm.c updated
bootadm ba_path->module
**********************************************************
______unchanged_portion_omitted_

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

 119 #define GRUB_DIR                "/boot/grub"
 120 #define GRUB_STAGE2             GRUB_DIR "/stage2"
 121 #define GRUB_MENU               "/boot/illumos.cfg"
 122 #define MENU_TMP                "/boot/illumos.cfg.tmp"
 121 #define GRUB_MENU               "/boot/grub/menu.lst"
 122 #define MENU_TMP                "/boot/grub/menu.lst.tmp"
 123 #define GRUB_BACKUP_MENU        "/etc/lu/GRUB_backup_menu"
 124 #define RAMDISK_SPECIAL         "/ramdisk"
 125 #define STUBBOOT                "/stubboot"
 126 #define MULTIBOOT               "/platform/i86pc/multiboot"
 127 #define GRUBSIGN_DIR            "/boot/grub/bootsign"
 128 #define GRUBSIGN_BACKUP         "/etc/bootsign"
 129 #define GRUBSIGN_UFS_PREFIX     "rootfs"
 130 #define GRUBSIGN_ZFS_PREFIX     "pool_"
 131 #define GRUBSIGN_LU_PREFIX      "BE_"
 132 #define UFS_SIGNATURE_LIST      "/var/run/grub_ufs_signatures"
 133 #define ZFS_LEGACY_MNTPT        "/tmp/bootadm_mnt_zfs_legacy"

 135 #define BOOTADM_RDONLY_TEST     "BOOTADM_RDONLY_TEST"

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

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

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

 157 #define INSTALLGRUB             "/sbin/installgrub"
 158 #define STAGE1                  "/boot/grub/stage1"
 159 #define STAGE2                  "/boot/grub/stage2"

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

 169 /*
```



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

 176 /*
 177  * Menu related
 178  * menu_cmd_t and menu_cmds must be kept in sync
 179  */
 180 char *menu_cmds[] = {
 181         "default_entry",/* DEFAULT_CMD */
 181         "default",      /* DEFAULT_CMD */
 182         "timeout",      /* TIMEOUT_CMD */
 183         "entry_name",   /* TITLE_CMD */
 184         "pool_uuid",    /* ROOT_CMD */
 185         "kernel_path$", /* KERNEL_CMD */
 186         "kernel_path",  /* KERNEL_DOLLAR_CMD */
 187         "module$",      /* MODULE_CMD */
 188         "module",       /* MODULE_DOLLAR_CMD */
 189         "=",            /* SEP_CMD */
 183         "title",        /* TITLE_CMD */
 184         "root",         /* ROOT_CMD */
 185         "kernel",       /* KERNEL_CMD */
 186         "kernel$",      /* KERNEL_DOLLAR_CMD */
 187         "module",       /* MODULE_CMD */
 188         "module$",      /* MODULE_DOLLAR_CMD */
 189         " ",            /* SEP_CMD */
 190         "#",            /* COMMENT_CMD */
 191         "chainloader",  /* CHAINLOADER_CMD */
 192         "args",         /* ARGS_CMD */
 193         "pool_label",   /* FINDROOT_CMD */
 194         "dataset",      /* BOOTFS_CMD */
 195         "kernel_options",/* KERNEL_OPTIONS_CMD */
 193         "findroot",     /* FINDROOT_CMD */
 194         "bootfs",       /* BOOTFS_CMD */
 196         NULL
 197 };
______unchanged_portion_omitted_

4679 int
4680 add_boot_entry(menu_t *mp,
4681         char *title,
4682         char *findroot,
4683         char *kernel,
4684         char *mod_kernel,
4685         char *module,
4686         char *bootfs)
4687 {
4688         int             lineNum;
4689         int             entryNum;
4690         char            linebuf[BAM_MAXLINE];
4691         menu_cmd_t      k_cmd;
4692         menu_cmd_t      m_cmd;
4693         const char      *fcn = "add_boot_entry()";
4694         char * options;
4695 #endif /* ! codereview */

4697         assert(mp);

4699         INJECT_ERROR1("ADD_BOOT_ENTRY_FINDROOT_NULL", findroot = NULL);
4700         if (findroot == NULL) {
4701                 bam_error(NULL_FINDROOT);
4702                 return (BAM_ERROR);
4703         }
```

```
4705         if (title == NULL) {
4706                 title = "Solaris";      /* default to Solaris */
4707         }
4708         if (kernel == NULL) {
4709                 bam_error(SUBOPT_MISS, menu_cmds[KERNEL_CMD]);
4710                 return (BAM_ERROR);
4711         }
4712         if (module == NULL) {
4713                 if (bam_direct != BAM_DIRECT_DBOOT) {
4714                         bam_error(SUBOPT_MISS, menu_cmds[MODULE_CMD]);
4715                         return (BAM_ERROR);
4716                 }

4718                 /* Figure the commands out from the kernel line */
4719                 if (strstr(kernel, "$ISADIR") != NULL) {
4720                         module = DIRECT_BOOT_ARCHIVE;
4721                 } else if (strstr(kernel, "amd64") != NULL) {
4722                         module = DIRECT_BOOT_ARCHIVE_64;
4723                 } else {
4724                         module = DIRECT_BOOT_ARCHIVE_32;
4725                 }
4726         }

4728         k_cmd = KERNEL_DOLLAR_CMD;
4729         m_cmd = MODULE_DOLLAR_CMD;

4731         if (mp->start) {
4732                 lineNum = mp->end->lineNum;
4733                 entryNum = mp->end->entryNum;
4734         } else {
4735                 lineNum = LINE_INIT;
4736                 entryNum = ENTRY_INIT;
4737         }

4739         /*
4740          * No separator for comment (HDR/FTR) commands
4741          * The syntax for comments is #<comment>
4742          */
4743         (void) snprintf(linebuf, sizeof (linebuf), "%s%s",
4744             menu_cmds[COMMENT_CMD], BAM_BOOTADM_HDR);
4745         line_parser(mp, linebuf, &lineNum, &entryNum);

4747         (void) snprintf(linebuf, sizeof (linebuf), "%s%s%s",
4748             menu_cmds[TITLE_CMD], menu_cmds[SEP_CMD], title);
4749         line_parser(mp, linebuf, &lineNum, &entryNum);

4751         (void) snprintf(linebuf, sizeof (linebuf), "%s%s%s",
4752             menu_cmds[FINDROOT_CMD], menu_cmds[SEP_CMD], findroot);
4753         line_parser(mp, linebuf, &lineNum, &entryNum);
4754         BAM_DPRINTF((D_ADD_FINDROOT_NUM, fcn, lineNum, entryNum));

4756         if (bootfs != NULL) {
4757                 (void) snprintf(linebuf, sizeof (linebuf), "%s%s%s",
4758                     menu_cmds[BOOTFS_CMD], menu_cmds[SEP_CMD], bootfs);
4759                 line_parser(mp, linebuf, &lineNum, &entryNum);
4760         }

4762         options = strpbrk(kernel, " \t");
4763         if (options)
4764                 *options++ = 0;

4766 #endif /* ! codereview */
4767         (void) snprintf(linebuf, sizeof (linebuf), "%s%s%s",
4768             menu_cmds[k_cmd], menu_cmds[SEP_CMD], kernel);
4769         line_parser(mp, linebuf, &lineNum, &entryNum);
```



```
4771         if (options) {
4772                 (void) snprintf(linebuf, sizeof (linebuf), "%s%s%s",
4773                     menu_cmds[KERNEL_OPTIONS_CMD], menu_cmds[SEP_CMD], options);
4774                 line_parser(mp, linebuf, &lineNum, &entryNum);
4775         }

4777 #endif /* ! codereview */
4778         if (mod_kernel != NULL) {
4779                 (void) snprintf(linebuf, sizeof (linebuf), "%s%s%s",
4780                     menu_cmds[m_cmd], menu_cmds[SEP_CMD], mod_kernel);
4781                 line_parser(mp, linebuf, &lineNum, &entryNum);
4782         }

4784         (void) snprintf(linebuf, sizeof (linebuf), "%s%s%s",
4785             menu_cmds[m_cmd], menu_cmds[SEP_CMD], module);
4786         line_parser(mp, linebuf, &lineNum, &entryNum);

4788         (void) snprintf(linebuf, sizeof (linebuf), "%s%s",
4789             menu_cmds[COMMENT_CMD], BAM_BOOTADM_FTR);
4790         line_parser(mp, linebuf, &lineNum, &entryNum);

4792         return (entryNum);
4793 }

4795 error_t
4796 delete_boot_entry(menu_t *mp, int entryNum, int quiet)
4797 {
4798         line_t          *lp;
4799         line_t          *freed;
4800         entry_t         *ent;
4801         entry_t         *tmp;
4802         int             deleted = 0;
4803         const char      *fcn = "delete_boot_entry()";

4805         assert(entryNum != ENTRY_INIT);

4807         tmp = NULL;

4809         ent = mp->entries;
4810         while (ent) {
4811                 lp = ent->start;

4813                 /*
4814                  * Check entry number and make sure it's a modifiable entry.
4815                  *
4816                  * Guidelines:
4817                  *      + We can modify a bootadm-created entry
4818                  *      + We can modify a libbe-created entry
4819                  */
4820                 if ((lp->flags != BAM_COMMENT &&
4821                     (((ent->flags & BAM_ENTRY_LIBBE) == 0) &&
4822                     strcmp(lp->arg, BAM_BOOTADM_HDR) != 0)) ||
4823                     (entryNum != ALL_ENTRIES && lp->entryNum != entryNum)) {
4824                         ent = ent->next;
4825                         continue;
4826                 }

4828                 /* free the entry content */
4829                 do {
4830                         freed = lp;
4831                         lp = lp->next;  /* prev stays the same */
4832                         BAM_DPRINTF((D_FREEING_LINE, fcn, freed->lineNum));
4833                         unlink_line(mp, freed);
4834                         line_free(freed);
4835                 } while (freed != ent->end);
```

```
4837                 /* free the entry_t structure */
4838                 assert(tmp == NULL);
4839                 tmp = ent;
4840                 ent = ent->next;
4841                 if (tmp->prev)
4842                         tmp->prev->next = ent;
4843                 else
4844                         mp->entries = ent;
4845                 if (ent)
4846                         ent->prev = tmp->prev;
4847                 BAM_DPRINTF((D_FREEING_ENTRY, fcn, tmp->entryNum));
4848                 free(tmp);
4849                 tmp = NULL;
4850                 deleted = 1;
4851         }

4853         assert(tmp == NULL);

4855         if (!deleted && entryNum != ALL_ENTRIES) {
4856                 if (quiet == DBE_PRINTERR)
4857                         bam_error(NO_BOOTADM_MATCH);
4858                 return (BAM_ERROR);
4859         }

4861         /*
4862          * Now that we have deleted an entry, update
4863          * the entry numbering and the default cmd.
4864          */
4865         update_numbering(mp);

4867         return (BAM_SUCCESS);
4868 }

4870 static error_t
4871 delete_all_entries(menu_t *mp, char *dummy, char *opt)
4872 {
4873         assert(mp);
4874         assert(dummy == NULL);
4875         assert(opt == NULL);

4877         BAM_DPRINTF((D_FUNC_ENTRY0, "delete_all_entries"));

4879         if (mp->start == NULL) {
4880                 bam_print(EMPTY_MENU);
4881                 return (BAM_SUCCESS);
4882         }

4884         if (delete_boot_entry(mp, ALL_ENTRIES, DBE_PRINTERR) != BAM_SUCCESS) {
4885                 return (BAM_ERROR);
4886         }

4888         return (BAM_WRITE);
4889 }

4891 static FILE *
4892 create_diskmap(char *osroot)
4893 {
4894         FILE *fp;
4895         char cmd[PATH_MAX + 16];
4896         char path[PATH_MAX];
4897         const char *fcn = "create_diskmap()";

4899         /* make sure we have a map file */
4900         fp = fopen(GRUBDISK_MAP, "r");
4901         if (fp == NULL) {
4902                 int     ret;
```



```
4904                 ret = snprintf(path, sizeof (path), "%s/%s", osroot,
4905                     CREATE_DISKMAP);
4906                 if (ret >= sizeof (path)) {
4907                         bam_error(PATH_TOO_LONG, osroot);
4908                         return (NULL);
4909                 }
4910                 if (is_safe_exec(path) == BAM_ERROR)
4911                         return (NULL);

4913                 (void) snprintf(cmd, sizeof (cmd),
4914                     "%s/%s > /dev/null", osroot, CREATE_DISKMAP);
4915                 if (exec_cmd(cmd, NULL) != 0)
4916                         return (NULL);
4917                 fp = fopen(GRUBDISK_MAP, "r");
4918                 INJECT_ERROR1("DISKMAP_CREATE_FAIL", fp = NULL);
4919                 if (fp) {
4920                         BAM_DPRINTF((D_CREATED_DISKMAP, fcn, GRUBDISK_MAP));
4921                 } else {
4922                         BAM_DPRINTF((D_CREATE_DISKMAP_FAIL, fcn, GRUBDISK_MAP));
4923                 }
4924         }
4925         return (fp);
4926 }

4928 #define SECTOR_SIZE     512

4930 static int
4931 get_partition(char *device)
4932 {
4933         int i, fd, is_pcfs, partno = -1;
4934         struct mboot *mboot;
4935         char boot_sect[SECTOR_SIZE];
4936         char *wholedisk, *slice;
4937 #ifdef i386
4938         ext_part_t *epp;
4939         uint32_t secnum, numsec;
4940         int rval, pno, ext_partno = -1;
4941 #endif

4943         /* form whole disk (p0) */
4944         slice = device + strlen(device) - 2;
4945         is_pcfs = (*slice != 's');
4946         if (!is_pcfs)
4947                 *slice = '\0';
4948         wholedisk = s_calloc(1, strlen(device) + 3);
4949         (void) snprintf(wholedisk, strlen(device) + 3, "%sp0", device);
4950         if (!is_pcfs)
4951                 *slice = 's';

4953         /* read boot sector */
4954         fd = open(wholedisk, O_RDONLY);
4955         if (fd == -1 || read(fd, boot_sect, SECTOR_SIZE) != SECTOR_SIZE) {
4956                 return (partno);
4957         }
4958         (void) close(fd);

4960 #ifdef i386
4961         /* Read/Initialize extended partition information */
4962         if ((rval = libfdisk_init(&epp, wholedisk, NULL, FDISK_READ_DISK))
4963             != FDISK_SUCCESS) {
4964                 switch (rval) {
4965                         /*
4966                          * FDISK_EBADLOGDRIVE and FDISK_ENOLOGDRIVE can
4967                          * be considered as soft errors and hence
4968                          * we do not return
```

```
4969                         */
4970                        case FDISK_EBADLOGDRIVE:
4971                                break;
4972                        case FDISK_ENOLOGDRIVE:
4973                                break;
4974                        case FDISK_EBADMAGIC:
4975                                /*FALLTHROUGH*/
4976                        default:
4977                                free(wholedisk);
4978                                libfdisk_fini(&epp);
4979                                return (partno);
4980                }
4981        }
4982 #endif
4983        free(wholedisk);

4985        /* parse fdisk table */
4986        mboot = (struct mboot *)((void *)boot_sect);
4987        for (i = 0; i < FD_NUMPART; i++) {
4988                struct ipart *part =
4989                    (struct ipart *)(uintptr_t)mboot->parts + i;
4990                if (is_pcfs) {  /* looking for solaris boot part */
4991                        if (part->systid == 0xbe) {
4992                                partno = i;
4993                                break;
4994                        }
4995                } else {        /* look for solaris partition, old and new */
4996 #ifdef i386
4997                        if ((part->systid == SUNIXOS &&
4998                            (fdisk_is_linux_swap(epp, part->relsect,
4999                            NULL) != 0)) || part->systid == SUNIXOS2) {
5000 #else
5001                        if (part->systid == SUNIXOS ||
5002                            part->systid == SUNIXOS2) {
5003 #endif
5004                                partno = i;
5005                                break;
5006                        }

5008 #ifdef i386
5009                        if (fdisk_is_dos_extended(part->systid))
5010                                ext_partno = i;
5011 #endif
5012                }
5013        }
5014 #ifdef i386
5015        /* If no primary solaris partition, check extended partition */
5016        if ((partno == -1) && (ext_partno != -1)) {
5017                rval = fdisk_get_solaris_part(epp, &pno, &secnum, &numsec);
5018                if (rval == FDISK_SUCCESS) {
5019                        partno = pno - 1;
5020                }
5021        }
5022        libfdisk_fini(&epp);
5023 #endif
5024        return (partno);
5025 }

5027 char *
5028 get_grubroot(char *osroot, char *osdev, char *menu_root)
5029 {
5030        char            *grubroot;      /* (hd#,#,#) */
5031        char            *slice;
5032        char            *grubhd;
5033        int             fdiskpart;
5034        int             found = 0;
```



```
5035        char            *devname;
5036        char            *ctdname = strstr(osdev, "dsk/");
5037        char            linebuf[PATH_MAX];
5038        FILE            *fp;

5040        INJECT_ERROR1("GRUBROOT_INVALID_OSDEV", ctdname = NULL);
5041        if (ctdname == NULL) {
5042                bam_error(INVALID_DEV_DSK, osdev);
5043                return (NULL);
5044        }

5046        if (menu_root && !menu_on_bootdisk(osroot, menu_root)) {
5047                /* menu bears no resemblance to our reality */
5048                bam_error(CANNOT_GRUBROOT_BOOTDISK, osdev);
5049                return (NULL);
5050        }

5052        ctdname += strlen("dsk/");
5053        slice = strrchr(ctdname, 's');
5054        if (slice)
5055                *slice = '\0';

5057        fp = create_diskmap(osroot);
5058        if (fp == NULL) {
5059                bam_error(DISKMAP_FAIL, osroot);
5060                return (NULL);
5061        }

5063        rewind(fp);
5064        while (s_fgets(linebuf, sizeof (linebuf), fp) != NULL) {
5065                grubhd = strtok(linebuf, " \t\n");
5066                if (grubhd)
5067                        devname = strtok(NULL, " \t\n");
5068                else
5069                        devname = NULL;
5070                if (devname && strcmp(devname, ctdname) == 0) {
5071                        found = 1;
5072                        break;
5073                }
5074        }

5076        if (slice)
5077                *slice = 's';

5079        (void) fclose(fp);
5080        fp = NULL;

5082        INJECT_ERROR1("GRUBROOT_BIOSDEV_FAIL", found = 0);
5083        if (found == 0) {
5084                bam_error(BIOSDEV_SKIP, osdev);
5085                return (NULL);
5086        }

5088        fdiskpart = get_partition(osdev);
5089        INJECT_ERROR1("GRUBROOT_FDISK_FAIL", fdiskpart = -1);
5090        if (fdiskpart == -1) {
5091                bam_error(FDISKPART_FAIL, osdev);
5092                return (NULL);
5093        }

5095        grubroot = s_calloc(1, 10);
5096        if (slice) {
5097                (void) snprintf(grubroot, 10, "(hd%s,%d,%c)",
5098                    grubhd, fdiskpart, slice[1] + 'a' - '0');
5099        } else
5100                (void) snprintf(grubroot, 10, "(hd%s,%d)",
```

```
5101                    grubhd, fdiskpart);

5103        assert(fp == NULL);
5104        assert(strncmp(grubroot, "(hd", strlen("(hd")) == 0);
5105        return (grubroot);
5106 }

5108 static char *
5109 find_primary_common(char *mntpt, char *fstype)
5110 {
5111        char            signdir[PATH_MAX];
5112        char            tmpsign[MAXNAMELEN + 1];
5113        char            *lu;
5114        char            *ufs;
5115        char            *zfs;
5116        DIR             *dirp = NULL;
5117        struct dirent   *entp;
5118        struct stat     sb;
5119        const char      *fcn = "find_primary_common()";

5121        (void) snprintf(signdir, sizeof (signdir), "%s/%s",
5122            mntpt, GRUBSIGN_DIR);

5124        if (stat(signdir, &sb) == -1) {
5125                BAM_DPRINTF((D_NO_SIGNDIR, fcn, signdir));
5126                return (NULL);
5127        }

5129        dirp = opendir(signdir);
5130        INJECT_ERROR1("SIGNDIR_OPENDIR_FAIL", dirp = NULL);
5131        if (dirp == NULL) {
5132                bam_error(OPENDIR_FAILED, signdir, strerror(errno));
5133                return (NULL);
5134        }

5136        ufs = zfs = lu = NULL;

5138        while (entp = readdir(dirp)) {
5139                if (strcmp(entp->d_name, ".") == 0 ||
5140                    strcmp(entp->d_name, "..") == 0)
5141                        continue;

5143                (void) snprintf(tmpsign, sizeof (tmpsign), "%s", entp->d_name);

5145                if (lu == NULL &&
5146                    strncmp(tmpsign, GRUBSIGN_LU_PREFIX,
5147                    strlen(GRUBSIGN_LU_PREFIX)) == 0) {
5148                        lu = s_strdup(tmpsign);
5149                }

5151                if (ufs == NULL &&
5152                    strncmp(tmpsign, GRUBSIGN_UFS_PREFIX,
5153                    strlen(GRUBSIGN_UFS_PREFIX)) == 0) {
5154                        ufs = s_strdup(tmpsign);
5155                }

5157                if (zfs == NULL &&
5158                    strncmp(tmpsign, GRUBSIGN_ZFS_PREFIX,
5159                    strlen(GRUBSIGN_ZFS_PREFIX)) == 0) {
5160                        zfs = s_strdup(tmpsign);
5161                }
5162        }

5164        BAM_DPRINTF((D_EXIST_PRIMARY_SIGNS, fcn,
5165            zfs ? zfs : "NULL",
5166            ufs ? ufs : "NULL",
```



```
5167            lu ? lu : "NULL"));

5169        if (dirp) {
5170                (void) closedir(dirp);
5171                dirp = NULL;
5172        }

5174        if (strcmp(fstype, "ufs") == 0 && zfs) {
5175                bam_error(SIGN_FSTYPE_MISMATCH, zfs, "ufs");
5176                free(zfs);
5177                zfs = NULL;
5178        } else if (strcmp(fstype, "zfs") == 0 && ufs) {
5179                bam_error(SIGN_FSTYPE_MISMATCH, ufs, "zfs");
5180                free(ufs);
5181                ufs = NULL;
5182        }

5184        assert(dirp == NULL);

5186        /* For now, we let Live Upgrade take care of its signature itself */
5187        if (lu) {
5188                BAM_DPRINTF((D_FREEING_LU_SIGNS, fcn, lu));
5189                free(lu);
5190                lu = NULL;
5191        }

5193        return (zfs ? zfs : ufs);
5194 }

5196 static char *
5197 find_backup_common(char *mntpt, char *fstype)
5198 {
5199        FILE            *bfp = NULL;
5200        char            tmpsign[MAXNAMELEN + 1];
5201        char            backup[PATH_MAX];
5202        char            *ufs;
5203        char            *zfs;
5204        char            *lu;
5205        int             error;
5206        const char      *fcn = "find_backup_common()";

5208        /*
5209         * We didn't find it in the primary directory.
5210         * Look at the backup
5211         */
5212        (void) snprintf(backup, sizeof (backup), "%s%s",
5213            mntpt, GRUBSIGN_BACKUP);

5215        bfp = fopen(backup, "r");
5216        if (bfp == NULL) {
5217                error = errno;
5218                if (bam_verbose) {
5219                        bam_error(OPEN_FAIL, backup, strerror(error));
5220                }
5221                BAM_DPRINTF((D_OPEN_FAIL, fcn, backup, strerror(error)));
5222                return (NULL);
5223        }

5225        ufs = zfs = lu = NULL;

5227        while (s_fgets(tmpsign, sizeof (tmpsign), bfp) != NULL) {

5229                if (lu == NULL &&
5230                    strncmp(tmpsign, GRUBSIGN_LU_PREFIX,
5231                    strlen(GRUBSIGN_LU_PREFIX)) == 0) {
5232                        lu = s_strdup(tmpsign);
```

```
5233                 }
5235                 if (ufs == NULL &&
5236                     strncmp(tmpsign, GRUBSIGN_UFS_PREFIX,
5237                     strlen(GRUBSIGN_UFS_PREFIX)) == 0) {
5238                         ufs = s_strdup(tmpsign);
5239                 }
5241                 if (zfs == NULL &&
5242                     strncmp(tmpsign, GRUBSIGN_ZFS_PREFIX,
5243                     strlen(GRUBSIGN_ZFS_PREFIX)) == 0) {
5244                         zfs = s_strdup(tmpsign);
5245                 }
5246         }
5248         BAM_DPRINTF((D_EXIST_BACKUP_SIGNS, fcn,
5249             zfs ? zfs : "NULL",
5250             ufs ? ufs : "NULL",
5251             lu ? lu : "NULL"));
5253         if (bfp) {
5254                 (void) fclose(bfp);
5255                 bfp = NULL;
5256         }
5258         if (strcmp(fstype, "ufs") == 0 && zfs) {
5259                 bam_error(SIGN_FSTYPE_MISMATCH, zfs, "ufs");
5260                 free(zfs);
5261                 zfs = NULL;
5262         } else if (strcmp(fstype, "zfs") == 0 && ufs) {
5263                 bam_error(SIGN_FSTYPE_MISMATCH, ufs, "zfs");
5264                 free(ufs);
5265                 ufs = NULL;
5266         }
5268         assert(bfp == NULL);
5270         /* For now, we let Live Upgrade take care of its signature itself */
5271         if (lu) {
5272                 BAM_DPRINTF((D_FREEING_LU_SIGNS, fcn, lu));
5273                 free(lu);
5274                 lu = NULL;
5275         }
5277         return (zfs ? zfs : ufs);
5278 }
5280 static char *
5281 find_ufs_existing(char *osroot)
5282 {
5283         char            *sign;
5284         const char      *fcn = "find_ufs_existing()";
5286         sign = find_primary_common(osroot, "ufs");
5287         if (sign == NULL) {
5288                 sign = find_backup_common(osroot, "ufs");
5289                 BAM_DPRINTF((D_EXIST_BACKUP_SIGN, fcn, sign ? sign : "NULL"));
5290         } else {
5291                 BAM_DPRINTF((D_EXIST_PRIMARY_SIGN, fcn, sign));
5292         }
5294         return (sign);
5295 }
5297 char *
5298 get_mountpoint(char *special, char *fstype)
```



```
5299 {
5300         FILE            *mntfp;
5301         struct mnttab   mp = {0};
5302         struct mnttab   mpref = {0};
5303         int             error;
5304         int             ret;
5305         const char      *fcn = "get_mountpoint()";
5307         BAM_DPRINTF((D_FUNC_ENTRY2, fcn, special, fstype));
5309         mntfp = fopen(MNTTAB, "r");
5310         error = errno;
5311         INJECT_ERROR1("MNTTAB_ERR_GET_MNTPT", mntfp = NULL);
5312         if (mntfp == NULL) {
5313                 bam_error(OPEN_FAIL, MNTTAB, strerror(error));
5314                 return (NULL);
5315         }
5317         mpref.mnt_special = special;
5318         mpref.mnt_fstype = fstype;
5320         ret = getmntany(mntfp, &mp, &mpref);
5321         INJECT_ERROR1("GET_MOUNTPOINT_MNTANY", ret = 1);
5322         if (ret != 0) {
5323                 (void) fclose(mntfp);
5324                 BAM_DPRINTF((D_NO_MNTPT, fcn, special, fstype));
5325                 return (NULL);
5326         }
5327         (void) fclose(mntfp);
5329         assert(mp.mnt_mountp);
5331         BAM_DPRINTF((D_GET_MOUNTPOINT_RET, fcn, special, mp.mnt_mountp));
5333         return (s_strdup(mp.mnt_mountp));
5334 }
5336 /*
5337  * Mounts a "legacy" top dataset (if needed)
5338  * Returns:     The mountpoint of the legacy top dataset or NULL on error
5339  *              mnted returns one of the above values defined for zfs_mnted_t
5340  */
5341 static char *
5342 mount_legacy_dataset(char *pool, zfs_mnted_t *mnted)
5343 {
5344         char            cmd[PATH_MAX];
5345         char            tmpmnt[PATH_MAX];
5346         filelist_t      flist = {0};
5347         char            *is_mounted;
5348         struct stat     sb;
5349         int             ret;
5350         const char      *fcn = "mount_legacy_dataset()";
5352         BAM_DPRINTF((D_FUNC_ENTRY1, fcn, pool));
5354         *mnted = ZFS_MNT_ERROR;
5356         (void) snprintf(cmd, sizeof (cmd),
5357             "/sbin/zfs get -Ho value mounted %s",
5358             pool);
5360         ret = exec_cmd(cmd, &flist);
5361         INJECT_ERROR1("Z_MOUNT_LEG_GET_MOUNTED_CMD", ret = 1);
5362         if (ret != 0) {
5363                 bam_error(ZFS_MNTED_FAILED, pool);
5364                 return (NULL);
```

```
5365         }

5367         INJECT_ERROR1("Z_MOUNT_LEG_GET_MOUNTED_OUT", flist.head = NULL);
5368         if ((flist.head == NULL) || (flist.head != flist.tail)) {
5369                 bam_error(BAD_ZFS_MNTED, pool);
5370                 filelist_free(&flist);
5371                 return (NULL);
5372         }

5374         is_mounted = strtok(flist.head->line, " \t\n");
5375         INJECT_ERROR1("Z_MOUNT_LEG_GET_MOUNTED_STRTOK_YES", is_mounted = "yes");
5376         INJECT_ERROR1("Z_MOUNT_LEG_GET_MOUNTED_STRTOK_NO", is_mounted = "no");
5377         if (strcmp(is_mounted, "no") != 0) {
5378                 filelist_free(&flist);
5379                 *mnted = LEGACY_ALREADY;
5380                 /* get_mountpoint returns a strdup'ed string */
5381                 BAM_DPRINTF((D_Z_MOUNT_TOP_LEG_ALREADY, fcn, pool));
5382                 return (get_mountpoint(pool, "zfs"));
5383         }

5385         filelist_free(&flist);

5387         /*
5388          * legacy top dataset is not mounted. Mount it now
5389          * First create a mountpoint.
5390          */
5391         (void) snprintf(tmpmnt, sizeof (tmpmnt), "%s.%d",
5392             ZFS_LEGACY_MNTPT, getpid());

5394         ret = stat(tmpmnt, &sb);
5395         if (ret == -1) {
5396                 BAM_DPRINTF((D_Z_MOUNT_TOP_LEG_MNTPT_ABS, fcn, pool, tmpmnt));
5397                 ret = mkdirp(tmpmnt, DIR_PERMS);
5398                 INJECT_ERROR1("Z_MOUNT_TOP_LEG_MNTPT_MKDIRP", ret = -1);
5399                 if (ret == -1) {
5400                         bam_error(MKDIR_FAILED, tmpmnt, strerror(errno));
5401                         return (NULL);
5402                 }
5403         } else {
5404                 BAM_DPRINTF((D_Z_MOUNT_TOP_LEG_MNTPT_PRES, fcn, pool, tmpmnt));
5405         }

5407         (void) snprintf(cmd, sizeof (cmd),
5408             "/sbin/mount -F zfs %s %s",
5409             pool, tmpmnt);

5411         ret = exec_cmd(cmd, NULL);
5412         INJECT_ERROR1("Z_MOUNT_TOP_LEG_MOUNT_CMD", ret = 1);
5413         if (ret != 0) {
5414                 bam_error(ZFS_MOUNT_FAILED, pool);
5415                 (void) rmdir(tmpmnt);
5416                 return (NULL);
5417         }

5419         *mnted = LEGACY_MOUNTED;
5420         BAM_DPRINTF((D_Z_MOUNT_TOP_LEG_MOUNTED, fcn, pool, tmpmnt));
5421         return (s_strdup(tmpmnt));
5422 }

5424 /*
5425  * Mounts the top dataset (if needed)
5426  * Returns:     The mountpoint of the top dataset or NULL on error
5427  *              mnted returns one of the above values defined for zfs_mnted_t
5428  */
5429 static char *
5430 mount_top_dataset(char *pool, zfs_mnted_t *mnted)
```



```
5431 {
5432         char            cmd[PATH_MAX];
5433         filelist_t      flist = {0};
5434         char            *is_mounted;
5435         char            *mntpt;
5436         char            *zmntpt;
5437         int             ret;
5438         const char      *fcn = "mount_top_dataset()";

5440         *mnted = ZFS_MNT_ERROR;

5442         BAM_DPRINTF((D_FUNC_ENTRY1, fcn, pool));

5444         /*
5445          * First check if the top dataset is a "legacy" dataset
5446          */
5447         (void) snprintf(cmd, sizeof (cmd),
5448             "/sbin/zfs get -Ho value mountpoint %s",
5449             pool);
5450         ret = exec_cmd(cmd, &flist);
5451         INJECT_ERROR1("Z_MOUNT_TOP_GET_MNTPT", ret = 1);
5452         if (ret != 0) {
5453                 bam_error(ZFS_MNTPT_FAILED, pool);
5454                 return (NULL);
5455         }

5457         if (flist.head && (flist.head == flist.tail)) {
5458                 char *legacy = strtok(flist.head->line, " \t\n");
5459                 if (legacy && strcmp(legacy, "legacy") == 0) {
5460                         filelist_free(&flist);
5461                         BAM_DPRINTF((D_Z_IS_LEGACY, fcn, pool));
5462                         return (mount_legacy_dataset(pool, mnted));
5463                 }
5464         }

5466         filelist_free(&flist);

5468         BAM_DPRINTF((D_Z_IS_NOT_LEGACY, fcn, pool));

5470         (void) snprintf(cmd, sizeof (cmd),
5471             "/sbin/zfs get -Ho value mounted %s",
5472             pool);

5474         ret = exec_cmd(cmd, &flist);
5475         INJECT_ERROR1("Z_MOUNT_TOP_NONLEG_GET_MOUNTED", ret = 1);
5476         if (ret != 0) {
5477                 bam_error(ZFS_MNTED_FAILED, pool);
5478                 return (NULL);
5479         }

5481         INJECT_ERROR1("Z_MOUNT_TOP_NONLEG_GET_MOUNTED_VAL", flist.head = NULL);
5482         if ((flist.head == NULL) || (flist.head != flist.tail)) {
5483                 bam_error(BAD_ZFS_MNTED, pool);
5484                 filelist_free(&flist);
5485                 return (NULL);
5486         }

5488         is_mounted = strtok(flist.head->line, " \t\n");
5489         INJECT_ERROR1("Z_MOUNT_TOP_NONLEG_GET_MOUNTED_YES", is_mounted = "yes");
5490         INJECT_ERROR1("Z_MOUNT_TOP_NONLEG_GET_MOUNTED_NO", is_mounted = "no");
5491         if (strcmp(is_mounted, "no") != 0) {
5492                 filelist_free(&flist);
5493                 *mnted = ZFS_ALREADY;
5494                 BAM_DPRINTF((D_Z_MOUNT_TOP_NONLEG_MOUNTED_ALREADY, fcn, pool));
5495                 goto mounted;
5496         }
```

```
5498         filelist_free(&flist);
5499         BAM_DPRINTF((D_Z_MOUNT_TOP_NONLEG_MOUNTED_NOT_ALREADY, fcn, pool));

5501         /* top dataset is not mounted. Mount it now */
5502         (void) snprintf(cmd, sizeof (cmd),
5503             "/sbin/zfs mount %s", pool);
5504         ret = exec_cmd(cmd, NULL);
5505         INJECT_ERROR1("Z_MOUNT_TOP_NONLEG_MOUNT_CMD", ret = 1);
5506         if (ret != 0) {
5507                 bam_error(ZFS_MOUNT_FAILED, pool);
5508                 return (NULL);
5509         }
5510         *mnted = ZFS_MOUNTED;
5511         BAM_DPRINTF((D_Z_MOUNT_TOP_NONLEG_MOUNTED_NOW, fcn, pool));
5512         /*FALLTHRU*/
5513 mounted:
5514         /*
5515          * Now get the mountpoint
5516          */
5517         (void) snprintf(cmd, sizeof (cmd),
5518             "/sbin/zfs get -Ho value mountpoint %s",
5519             pool);

5521         ret = exec_cmd(cmd, &flist);
5522         INJECT_ERROR1("Z_MOUNT_TOP_NONLEG_GET_MNTPT_CMD", ret = 1);
5523         if (ret != 0) {
5524                 bam_error(ZFS_MNTPT_FAILED, pool);
5525                 goto error;
5526         }

5528         INJECT_ERROR1("Z_MOUNT_TOP_NONLEG_GET_MNTPT_OUT", flist.head = NULL);
5529         if ((flist.head == NULL) || (flist.head != flist.tail)) {
5530                 bam_error(NULL_ZFS_MNTPT, pool);
5531                 goto error;
5532         }

5534         mntpt = strtok(flist.head->line, " \t\n");
5535         INJECT_ERROR1("Z_MOUNT_TOP_NONLEG_GET_MNTPT_STRTOK", mntpt = "foo");
5536         if (*mntpt != '/') {
5537                 bam_error(BAD_ZFS_MNTPT, pool, mntpt);
5538                 goto error;
5539         }
5540         zmntpt = s_strdup(mntpt);

5542         filelist_free(&flist);

5544         BAM_DPRINTF((D_Z_MOUNT_TOP_NONLEG_MNTPT, fcn, pool, zmntpt));

5546         return (zmntpt);

5548 error:
5549         filelist_free(&flist);
5550         (void) umount_top_dataset(pool, *mnted, NULL);
5551         BAM_DPRINTF((D_RETURN_FAILURE, fcn));
5552         return (NULL);
5553 }

5555 static int
5556 umount_top_dataset(char *pool, zfs_mnted_t mnted, char *mntpt)
5557 {
5558         char            cmd[PATH_MAX];
5559         int             ret;
5560         const char      *fcn = "umount_top_dataset()";

5562         INJECT_ERROR1("Z_UMOUNT_TOP_INVALID_STATE", mnted = ZFS_MNT_ERROR);
```



```
5563         switch (mnted) {
5564         case LEGACY_ALREADY:
5565         case ZFS_ALREADY:
5566                 /* nothing to do */
5567                 BAM_DPRINTF((D_Z_UMOUNT_TOP_ALREADY_NOP, fcn, pool,
5568                     mntpt ? mntpt : "NULL"));
5569                 free(mntpt);
5570                 return (BAM_SUCCESS);
5571         case LEGACY_MOUNTED:
5572                 (void) snprintf(cmd, sizeof (cmd),
5573                     "/sbin/umount %s", pool);
5574                 ret = exec_cmd(cmd, NULL);
5575                 INJECT_ERROR1("Z_UMOUNT_TOP_LEGACY_UMOUNT_FAIL", ret = 1);
5576                 if (ret != 0) {
5577                         bam_error(UMOUNT_FAILED, pool);
5578                         free(mntpt);
5579                         return (BAM_ERROR);
5580                 }
5581                 if (mntpt)
5582                         (void) rmdir(mntpt);
5583                 free(mntpt);
5584                 BAM_DPRINTF((D_Z_UMOUNT_TOP_LEGACY, fcn, pool));
5585                 return (BAM_SUCCESS);
5586         case ZFS_MOUNTED:
5587                 free(mntpt);
5588                 (void) snprintf(cmd, sizeof (cmd),
5589                     "/sbin/zfs unmount %s", pool);
5590                 ret = exec_cmd(cmd, NULL);
5591                 INJECT_ERROR1("Z_UMOUNT_TOP_NONLEG_UMOUNT_FAIL", ret = 1);
5592                 if (ret != 0) {
5593                         bam_error(UMOUNT_FAILED, pool);
5594                         return (BAM_ERROR);
5595                 }
5596                 BAM_DPRINTF((D_Z_UMOUNT_TOP_NONLEG, fcn, pool));
5597                 return (BAM_SUCCESS);
5598         default:
5599                 bam_error(INT_BAD_MNTSTATE, pool);
5600                 return (BAM_ERROR);
5601         }
5602         /*NOTREACHED*/
5603 }

5605 /*
5606  * For ZFS, osdev can be one of two forms
5607  * It can be a "special" file as seen in mnttab: rpool/ROOT/szboot_0402
5608  * It can be a /dev/[r]dsk special file. We handle both instances
5609  */
5610 static char *
5611 get_pool(char *osdev)
5612 {
5613         char            cmd[PATH_MAX];
5614         char            buf[PATH_MAX];
5615         filelist_t      flist = {0};
5616         char            *pool;
5617         char            *cp;
5618         char            *slash;
5619         int             ret;
5620         const char      *fcn = "get_pool()";

5622         INJECT_ERROR1("GET_POOL_OSDEV", osdev = NULL);
5623         if (osdev == NULL) {
5624                 bam_error(GET_POOL_OSDEV_NULL);
5625                 return (NULL);
5626         }

5628         BAM_DPRINTF((D_GET_POOL_OSDEV, fcn, osdev));
```

```
5630         if (osdev[0] != '/') {
5631                 (void) strlcpy(buf, osdev, sizeof (buf));
5632                 slash = strchr(buf, '/');
5633                 if (slash)
5634                         *slash = '\0';
5635                 pool = s_strdup(buf);
5636                 BAM_DPRINTF((D_GET_POOL_RET, fcn, pool));
5637                 return (pool);
5638         } else if (strncmp(osdev, "/dev/dsk/", strlen("/dev/dsk/")) != 0 &&
5639             strncmp(osdev, "/dev/rdsk/", strlen("/dev/rdsk/")) != 0) {
5640                 bam_error(GET_POOL_BAD_OSDEV, osdev);
5641                 return (NULL);
5642         }
5643
5644         /*
5645          * Call the zfs fstyp directly since this is a zpool. This avoids
5646          * potential pcfs conflicts if the first block wasn't cleared.
5647          */
5648         (void) snprintf(cmd, sizeof (cmd),
5649             "/usr/lib/fs/zfs/fstyp -a %s 2>/dev/null | /bin/grep '^name:'",
5650             osdev);
5651
5652         ret = exec_cmd(cmd, &flist);
5653         INJECT_ERROR1("GET_POOL_FSTYP", ret = 1);
5654         if (ret != 0) {
5655                 bam_error(FSTYP_A_FAILED, osdev);
5656                 return (NULL);
5657         }
5658
5659         INJECT_ERROR1("GET_POOL_FSTYP_OUT", flist.head = NULL);
5660         if ((flist.head == NULL) || (flist.head != flist.tail)) {
5661                 bam_error(NULL_FSTYP_A, osdev);
5662                 filelist_free(&flist);
5663                 return (NULL);
5664         }
5665
5666         (void) strtok(flist.head->line, "'");
5667         cp = strtok(NULL, "'");
5668         INJECT_ERROR1("GET_POOL_FSTYP_STRTOK", cp = NULL);
5669         if (cp == NULL) {
5670                 bam_error(BAD_FSTYP_A, osdev);
5671                 filelist_free(&flist);
5672                 return (NULL);
5673         }
5674
5675         pool = s_strdup(cp);
5676
5677         filelist_free(&flist);
5678
5679         BAM_DPRINTF((D_GET_POOL_RET, fcn, pool));
5680
5681         return (pool);
5682 }
5683
5684 static char *
5685 find_zfs_existing(char *osdev)
5686 {
5687         char            *pool;
5688         zfs_mnted_t     mnted;
5689         char            *mntpt;
5690         char            *sign;
5691         const char      *fcn = "find_zfs_existing()";
5692
5693         pool = get_pool(osdev);
5694         INJECT_ERROR1("ZFS_FIND_EXIST_POOL", pool = NULL);
```



```
5695         if (pool == NULL) {
5696                 bam_error(ZFS_GET_POOL_FAILED, osdev);
5697                 return (NULL);
5698         }
5699
5700         mntpt = mount_top_dataset(pool, &mnted);
5701         INJECT_ERROR1("ZFS_FIND_EXIST_MOUNT_TOP", mntpt = NULL);
5702         if (mntpt == NULL) {
5703                 bam_error(ZFS_MOUNT_TOP_DATASET_FAILED, pool);
5704                 free(pool);
5705                 return (NULL);
5706         }
5707
5708         sign = find_primary_common(mntpt, "zfs");
5709         if (sign == NULL) {
5710                 sign = find_backup_common(mntpt, "zfs");
5711                 BAM_DPRINTF((D_EXIST_BACKUP_SIGN, fcn, sign ? sign : "NULL"));
5712         } else {
5713                 BAM_DPRINTF((D_EXIST_PRIMARY_SIGN, fcn, sign));
5714         }
5715
5716         (void) umount_top_dataset(pool, mnted, mntpt);
5717
5718         free(pool);
5719
5720         return (sign);
5721 }
5722
5723 static char *
5724 find_existing_sign(char *osroot, char *osdev, char *fstype)
5725 {
5726         const char              *fcn = "find_existing_sign()";
5727
5728         INJECT_ERROR1("FIND_EXIST_NOTSUP_FS", fstype = "foofs");
5729         if (strcmp(fstype, "ufs") == 0) {
5730                 BAM_DPRINTF((D_CHECK_UFS_EXIST_SIGN, fcn));
5731                 return (find_ufs_existing(osroot));
5732         } else if (strcmp(fstype, "zfs") == 0) {
5733                 BAM_DPRINTF((D_CHECK_ZFS_EXIST_SIGN, fcn));
5734                 return (find_zfs_existing(osdev));
5735         } else {
5736                 bam_error(GRUBSIGN_NOTSUP, fstype);
5737                 return (NULL);
5738         }
5739 }
5740
5741 #define MH_HASH_SZ      16
5742
5743 typedef enum {
5744         MH_ERROR = -1,
5745         MH_NOMATCH,
5746         MH_MATCH
5747 } mh_search_t;
5748
5749 typedef struct mcache {
5750         char    *mc_special;
5751         char    *mc_mntpt;
5752         char    *mc_fstype;
5753         struct mcache *mc_next;
5754 } mcache_t;
5755
5756 typedef struct mhash {
5757         mcache_t *mh_hash[MH_HASH_SZ];
5758 } mhash_t;
5759
5760 static int
```

```
5761 mhash_fcn(char *key)
5762 {
5763         int             i;
5764         uint64_t        sum = 0;

5766         for (i = 0; key[i] != '\0'; i++) {
5767                 sum += (uchar_t)key[i];
5768         }

5770         sum %= MH_HASH_SZ;

5772         assert(sum < MH_HASH_SZ);

5774         return (sum);
5775 }

5777 static mhash_t *
5778 cache_mnttab(void)
5779 {
5780         FILE            *mfp;
5781         struct extmnttab mnt;
5782         mcache_t        *mcp;
5783         mhash_t         *mhp;
5784         char            *ctds;
5785         int             idx;
5786         int             error;
5787         char            *special_dup;
5788         const char      *fcn = "cache_mnttab()";

5790         mfp = fopen(MNTTAB, "r");
5791         error = errno;
5792         INJECT_ERROR1("CACHE_MNTTAB_MNTTAB_ERR", mfp = NULL);
5793         if (mfp == NULL) {
5794                 bam_error(OPEN_FAIL, MNTTAB, strerror(error));
5795                 return (NULL);
5796         }

5798         mhp = s_calloc(1, sizeof (mhash_t));

5800         resetmnttab(mfp);

5802         while (getextmntent(mfp, &mnt, sizeof (mnt)) == 0) {
5803                 /* only cache ufs */
5804                 if (strcmp(mnt.mnt_fstype, "ufs") != 0)
5805                         continue;

5807                 /* basename() modifies its arg, so dup it */
5808                 special_dup = s_strdup(mnt.mnt_special);
5809                 ctds = basename(special_dup);

5811                 mcp = s_calloc(1, sizeof (mcache_t));
5812                 mcp->mc_special = s_strdup(ctds);
5813                 mcp->mc_mntpt = s_strdup(mnt.mnt_mountp);
5814                 mcp->mc_fstype = s_strdup(mnt.mnt_fstype);
5815                 BAM_DPRINTF((D_CACHE_MNTS, fcn, ctds,
5816                     mnt.mnt_mountp, mnt.mnt_fstype));
5817                 idx = mhash_fcn(ctds);
5818                 mcp->mc_next = mhp->mh_hash[idx];
5819                 mhp->mh_hash[idx] = mcp;
5820                 free(special_dup);
5821         }

5823         (void) fclose(mfp);

5825         return (mhp);
5826 }
```



```
5828 static void
5829 free_mnttab(mhash_t *mhp)
5830 {
5831         mcache_t        *mcp;
5832         int             i;

5834         for (i = 0; i < MH_HASH_SZ; i++) {
5835                 /*LINTED*/
5836                 while (mcp = mhp->mh_hash[i]) {
5837                         mhp->mh_hash[i] = mcp->mc_next;
5838                         free(mcp->mc_special);
5839                         free(mcp->mc_mntpt);
5840                         free(mcp->mc_fstype);
5841                         free(mcp);
5842                 }
5843         }

5845         for (i = 0; i < MH_HASH_SZ; i++) {
5846                 assert(mhp->mh_hash[i] == NULL);
5847         }
5848         free(mhp);
5849 }

5851 static mh_search_t
5852 search_hash(mhash_t *mhp, char *special, char **mntpt)
5853 {
5854         int             idx;
5855         mcache_t        *mcp;
5856         const char      *fcn = "search_hash()";

5858         assert(mntpt);

5860         *mntpt = NULL;

5862         INJECT_ERROR1("SEARCH_HASH_FULL_PATH", special = "/foo");
5863         if (strchr(special, '/')) {
5864                 bam_error(INVALID_MHASH_KEY, special);
5865                 return (MH_ERROR);
5866         }

5868         idx = mhash_fcn(special);

5870         for (mcp = mhp->mh_hash[idx]; mcp; mcp = mcp->mc_next) {
5871                 if (strcmp(mcp->mc_special, special) == 0)
5872                         break;
5873         }

5875         if (mcp == NULL) {
5876                 BAM_DPRINTF((D_MNTTAB_HASH_NOMATCH, fcn, special));
5877                 return (MH_NOMATCH);
5878         }

5880         assert(strcmp(mcp->mc_fstype, "ufs") == 0);
5881         *mntpt = mcp->mc_mntpt;
5882         BAM_DPRINTF((D_MNTTAB_HASH_MATCH, fcn, special));
5883         return (MH_MATCH);
5884 }

5886 static int
5887 check_add_ufs_sign_to_list(FILE *tfp, char *mntpt)
5888 {
5889         char            *sign;
5890         char            *signline;
5891         char            signbuf[MAXNAMELEN];
5892         int             len;
```

```
5893         int             error;
5894         const char      *fcn = "check_add_ufs_sign_to_list()";

5896         /* safe to specify NULL as "osdev" arg for UFS */
5897         sign = find_existing_sign(mntpt, NULL, "ufs");
5898         if (sign == NULL) {
5899                 /* No existing signature, nothing to add to list */
5900                 BAM_DPRINTF((D_NO_SIGN_TO_LIST, fcn, mntpt));
5901                 return (0);
5902         }

5904         (void) snprintf(signbuf, sizeof (signbuf), "%s\n", sign);
5905         signline = signbuf;

5907         INJECT_ERROR1("UFS_MNTPT_SIGN_NOTUFS", signline = "pool_rpool10\n");
5908         if (strncmp(signline, GRUBSIGN_UFS_PREFIX,
5909             strlen(GRUBSIGN_UFS_PREFIX))) {
5910                 bam_error(INVALID_UFS_SIGNATURE, sign);
5911                 free(sign);
5912                 /* ignore invalid signatures */
5913                 return (0);
5914         }

5916         len = fputs(signline, tfp);
5917         error = errno;
5918         INJECT_ERROR1("SIGN_LIST_PUTS_ERROR", len = 0);
5919         if (len != strlen(signline)) {
5920                 bam_error(SIGN_LIST_FPUTS_ERR, sign, strerror(error));
5921                 free(sign);
5922                 return (-1);
5923         }

5925         free(sign);

5927         BAM_DPRINTF((D_SIGN_LIST_PUTS_DONE, fcn, mntpt));
5928         return (0);
5929 }

5931 /*
5932  * slice is a basename not a full pathname
5933  */
5934 static int
5935 process_slice_common(char *slice, FILE *tfp, mhash_t *mhp, char *tmpmnt)
5936 {
5937         int             ret;
5938         char            cmd[PATH_MAX];
5939         char            path[PATH_MAX];
5940         struct stat     sbuf;
5941         char            *mntpt;
5942         filelist_t      flist = {0};
5943         char            *fstype;
5944         char            blkslice[PATH_MAX];
5945         const char      *fcn = "process_slice_common()";


5948         ret = search_hash(mhp, slice, &mntpt);
5949         switch (ret) {
5950                 case MH_MATCH:
5951                         if (check_add_ufs_sign_to_list(tfp, mntpt) == -1)
5952                                 return (-1);
5953                         else
5954                                 return (0);
5955                 case MH_NOMATCH:
5956                         break;
5957                 case MH_ERROR:
5958                 default:
```



```
5959                         return (-1);
5960         }

5962         (void) snprintf(path, sizeof (path), "/dev/rdsk/%s", slice);
5963         if (stat(path, &sbuf) == -1) {
5964                 BAM_DPRINTF((D_SLICE_ENOENT, fcn, path));
5965                 return (0);
5966         }

5968         /* Check if ufs. Call ufs fstyp directly to avoid pcfs conflicts. */
5969         (void) snprintf(cmd, sizeof (cmd),
5970             "/usr/lib/fs/ufs/fstyp /dev/rdsk/%s 2>/dev/null",
5971             slice);

5973         if (exec_cmd(cmd, &flist) != 0) {
5974                 if (bam_verbose)
5975                         bam_print(FSTYP_FAILED, slice);
5976                 return (0);
5977         }

5979         if ((flist.head == NULL) || (flist.head != flist.tail)) {
5980                 if (bam_verbose)
5981                         bam_print(FSTYP_BAD, slice);
5982                 filelist_free(&flist);
5983                 return (0);
5984         }

5986         fstype = strtok(flist.head->line, " \t\n");
5987         if (fstype == NULL || strcmp(fstype, "ufs") != 0) {
5988                 if (bam_verbose)
5989                         bam_print(NOT_UFS_SLICE, slice, fstype);
5990                 filelist_free(&flist);
5991                 return (0);
5992         }

5994         filelist_free(&flist);

5996         /*
5997          * Since we are mounting the filesystem read-only, the
5998          * the last mount field of the superblock is unchanged
5999          * and does not need to be fixed up post-mount;
6000          */

6002         (void) snprintf(blkslice, sizeof (blkslice), "/dev/dsk/%s",
6003             slice);

6005         (void) snprintf(cmd, sizeof (cmd),
6006             "/usr/sbin/mount -F ufs -o ro %s %s "
6007             "> /dev/null 2>&1", blkslice, tmpmnt);

6009         if (exec_cmd(cmd, NULL) != 0) {
6010                 if (bam_verbose)
6011                         bam_print(MOUNT_FAILED, blkslice, "ufs");
6012                 return (0);
6013         }

6015         ret = check_add_ufs_sign_to_list(tfp, tmpmnt);

6017         (void) snprintf(cmd, sizeof (cmd),
6018             "/usr/sbin/umount -f %s > /dev/null 2>&1",
6019             tmpmnt);

6021         if (exec_cmd(cmd, NULL) != 0) {
6022                 bam_print(UMOUNT_FAILED, slice);
6023                 return (0);
6024         }
```

```
6026         return (ret);
6027 }

6029 static int
6030 process_vtoc_slices(
6031         char *s0,
6032         struct vtoc *vtoc,
6033         FILE *tfp,
6034         mhash_t *mhp,
6035         char *tmpmnt)
6036 {
6037         int             idx;
6038         char            slice[PATH_MAX];
6039         size_t          len;
6040         char            *cp;
6041         const char      *fcn = "process_vtoc_slices()";

6043         len = strlen(s0);

6045         assert(s0[len - 2] == 's' && s0[len - 1] == '0');

6047         s0[len - 1] = '\0';

6049         (void) strlcpy(slice, s0, sizeof (slice));

6051         s0[len - 1] = '0';

6053         cp = slice + len - 1;

6055         for (idx = 0; idx < vtoc->v_nparts; idx++) {

6057                 (void) snprintf(cp, sizeof (slice) - (len - 1), "%u", idx);

6059                 if (vtoc->v_part[idx].p_size == 0) {
6060                         BAM_DPRINTF((D_VTOC_SIZE_ZERO, fcn, slice));
6061                         continue;
6062                 }

6064                 /* Skip "SWAP", "USR", "BACKUP", "VAR", "HOME", "ALTSCTR" */
6065                 switch (vtoc->v_part[idx].p_tag) {
6066                 case V_SWAP:
6067                 case V_USR:
6068                 case V_BACKUP:
6069                 case V_VAR:
6070                 case V_HOME:
6071                 case V_ALTSCTR:
6072                         BAM_DPRINTF((D_VTOC_NOT_ROOT_TAG, fcn, slice));
6073                         continue;
6074                 default:
6075                         BAM_DPRINTF((D_VTOC_ROOT_TAG, fcn, slice));
6076                         break;
6077                 }

6079                 /* skip unmountable and readonly slices */
6080                 switch (vtoc->v_part[idx].p_flag) {
6081                 case V_UNMNT:
6082                 case V_RONLY:
6083                         BAM_DPRINTF((D_VTOC_NOT_RDWR_FLAG, fcn, slice));
6084                         continue;
6085                 default:
6086                         BAM_DPRINTF((D_VTOC_RDWR_FLAG, fcn, slice));
6087                         break;
6088                 }

6090                 if (process_slice_common(slice, tfp, mhp, tmpmnt) == -1) {
```



```
6091                         return (-1);
6092                 }
6093         }

6095         return (0);
6096 }

6098 static int
6099 process_efi_slices(
6100         char *s0,
6101         struct dk_gpt *efi,
6102         FILE *tfp,
6103         mhash_t *mhp,
6104         char *tmpmnt)
6105 {
6106         int             idx;
6107         char            slice[PATH_MAX];
6108         size_t          len;
6109         char            *cp;
6110         const char      *fcn = "process_efi_slices()";

6112         len = strlen(s0);

6114         assert(s0[len - 2] == 's' && s0[len - 1] == '0');

6116         s0[len - 1] = '\0';

6118         (void) strlcpy(slice, s0, sizeof (slice));

6120         s0[len - 1] = '0';

6122         cp = slice + len - 1;

6124         for (idx = 0; idx < efi->efi_nparts; idx++) {

6126                 (void) snprintf(cp, sizeof (slice) - (len - 1), "%u", idx);

6128                 if (efi->efi_parts[idx].p_size == 0) {
6129                         BAM_DPRINTF((D_EFI_SIZE_ZERO, fcn, slice));
6130                         continue;
6131                 }

6133                 /* Skip "SWAP", "USR", "BACKUP", "VAR", "HOME", "ALTSCTR" */
6134                 switch (efi->efi_parts[idx].p_tag) {
6135                 case V_SWAP:
6136                 case V_USR:
6137                 case V_BACKUP:
6138                 case V_VAR:
6139                 case V_HOME:
6140                 case V_ALTSCTR:
6141                         BAM_DPRINTF((D_EFI_NOT_ROOT_TAG, fcn, slice));
6142                         continue;
6143                 default:
6144                         BAM_DPRINTF((D_EFI_ROOT_TAG, fcn, slice));
6145                         break;
6146                 }

6148                 /* skip unmountable and readonly slices */
6149                 switch (efi->efi_parts[idx].p_flag) {
6150                 case V_UNMNT:
6151                 case V_RONLY:
6152                         BAM_DPRINTF((D_EFI_NOT_RDWR_FLAG, fcn, slice));
6153                         continue;
6154                 default:
6155                         BAM_DPRINTF((D_EFI_RDWR_FLAG, fcn, slice));
6156                         break;
```

```
6157                }

6159                if (process_slice_common(slice, tfp, mhp, tmpmnt) == -1) {
6160                        return (-1);
6161                }
6162        }

6164        return (0);
6165 }

6167 /*
6168  * s0 is a basename not a full path
6169  */
6170 static int
6171 process_slice0(char *s0, FILE *tfp, mhash_t *mhp, char *tmpmnt)
6172 {
6173        struct vtoc             vtoc;
6174        struct dk_gpt           *efi;
6175        char                    s0path[PATH_MAX];
6176        struct stat             sbuf;
6177        int                     e_flag;
6178        int                     v_flag;
6179        int                     retval;
6180        int                     err;
6181        int                     fd;
6182        const char              *fcn = "process_slice0()";

6184        (void) snprintf(s0path, sizeof (s0path), "/dev/rdsk/%s", s0);

6186        if (stat(s0path, &sbuf) == -1) {
6187                BAM_DPRINTF((D_SLICE0_ENOENT, fcn, s0path));
6188                return (0);
6189        }

6191        fd = open(s0path, O_NONBLOCK|O_RDONLY);
6192        if (fd == -1) {
6193                bam_error(OPEN_FAIL, s0path, strerror(errno));
6194                return (0);
6195        }

6197        e_flag = v_flag = 0;
6198        retval = ((err = read_vtoc(fd, &vtoc)) >= 0) ? 0 : err;
6199        switch (retval) {
6200                case VT_EIO:
6201                        BAM_DPRINTF((D_VTOC_READ_FAIL, fcn, s0path));
6202                        break;
6203                case VT_EINVAL:
6204                        BAM_DPRINTF((D_VTOC_INVALID, fcn, s0path));
6205                        break;
6206                case VT_ERROR:
6207                        BAM_DPRINTF((D_VTOC_UNKNOWN_ERR, fcn, s0path));
6208                        break;
6209                case VT_ENOTSUP:
6210                        e_flag = 1;
6211                        BAM_DPRINTF((D_VTOC_NOTSUP, fcn, s0path));
6212                        break;
6213                case 0:
6214                        v_flag = 1;
6215                        BAM_DPRINTF((D_VTOC_READ_SUCCESS, fcn, s0path));
6216                        break;
6217                default:
6218                        BAM_DPRINTF((D_VTOC_UNKNOWN_RETCODE, fcn, s0path));
6219                        break;
6220        }
```



```
6223        if (e_flag) {
6224                e_flag = 0;
6225                retval = ((err = efi_alloc_and_read(fd, &efi)) >= 0) ? 0 : err;
6226                switch (retval) {
6227                case VT_EIO:
6228                        BAM_DPRINTF((D_EFI_READ_FAIL, fcn, s0path));
6229                        break;
6230                case VT_EINVAL:
6231                        BAM_DPRINTF((D_EFI_INVALID, fcn, s0path));
6232                        break;
6233                case VT_ERROR:
6234                        BAM_DPRINTF((D_EFI_UNKNOWN_ERR, fcn, s0path));
6235                        break;
6236                case VT_ENOTSUP:
6237                        BAM_DPRINTF((D_EFI_NOTSUP, fcn, s0path));
6238                        break;
6239                case 0:
6240                        e_flag = 1;
6241                        BAM_DPRINTF((D_EFI_READ_SUCCESS, fcn, s0path));
6242                        break;
6243                default:
6244                        BAM_DPRINTF((D_EFI_UNKNOWN_RETCODE, fcn, s0path));
6245                        break;
6246                }
6247        }

6249        (void) close(fd);

6251        if (v_flag) {
6252                retval = process_vtoc_slices(s0,
6253                    &vtoc, tfp, mhp, tmpmnt);
6254        } else if (e_flag) {
6255                retval = process_efi_slices(s0,
6256                    efi, tfp, mhp, tmpmnt);
6257        } else {
6258                BAM_DPRINTF((D_NOT_VTOC_OR_EFI, fcn, s0path));
6259                return (0);
6260        }

6262        return (retval);
6263 }

6265 /*
6266  * Find and create a list of all existing UFS boot signatures
6267  */
6268 static int
6269 FindAllUfsSignatures(void)
6270 {
6271        mhash_t         *mnttab_hash;
6272        DIR             *dirp = NULL;
6273        struct dirent   *dp;
6274        char            tmpmnt[PATH_MAX];
6275        char            cmd[PATH_MAX];
6276        struct stat     sb;
6277        int             fd;
6278        FILE            *tfp;
6279        size_t          len;
6280        int             ret;
6281        int             error;
6282        const char      *fcn = "FindAllUfsSignatures()";

6284        if (stat(UFS_SIGNATURE_LIST, &sb) != -1)  {
6285                bam_print(SIGNATURE_LIST_EXISTS, UFS_SIGNATURE_LIST);
6286                return (0);
6287        }
```

```
6289         fd = open(UFS_SIGNATURE_LIST".tmp",
6290             O_RDWR|O_CREAT|O_TRUNC, 0644);
6291         error = errno;
6292         INJECT_ERROR1("SIGN_LIST_TMP_TRUNC", fd = -1);
6293         if (fd == -1) {
6294                 bam_error(OPEN_FAIL, UFS_SIGNATURE_LIST".tmp", strerror(error));
6295                 return (-1);
6296         }

6298         ret = close(fd);
6299         error = errno;
6300         INJECT_ERROR1("SIGN_LIST_TMP_CLOSE", ret = -1);
6301         if (ret == -1) {
6302                 bam_error(CLOSE_FAIL, UFS_SIGNATURE_LIST".tmp",
6303                     strerror(error));
6304                 (void) unlink(UFS_SIGNATURE_LIST".tmp");
6305                 return (-1);
6306         }

6308         tfp = fopen(UFS_SIGNATURE_LIST".tmp", "a");
6309         error = errno;
6310         INJECT_ERROR1("SIGN_LIST_APPEND_FOPEN", tfp = NULL);
6311         if (tfp == NULL) {
6312                 bam_error(OPEN_FAIL, UFS_SIGNATURE_LIST".tmp", strerror(error));
6313                 (void) unlink(UFS_SIGNATURE_LIST".tmp");
6314                 return (-1);
6315         }

6317         mnttab_hash = cache_mnttab();
6318         INJECT_ERROR1("CACHE_MNTTAB_ERROR", mnttab_hash = NULL);
6319         if (mnttab_hash == NULL) {
6320                 (void) fclose(tfp);
6321                 (void) unlink(UFS_SIGNATURE_LIST".tmp");
6322                 bam_error(CACHE_MNTTAB_FAIL, fcn);
6323                 return (-1);
6324         }

6326         (void) snprintf(tmpmnt, sizeof (tmpmnt),
6327             "/tmp/bootadm_ufs_sign_mnt.%d", getpid());
6328         (void) unlink(tmpmnt);

6330         ret = mkdirp(tmpmnt, DIR_PERMS);
6331         error = errno;
6332         INJECT_ERROR1("MKDIRP_SIGN_MNT", ret = -1);
6333         if (ret == -1) {
6334                 bam_error(MKDIR_FAILED, tmpmnt, strerror(error));
6335                 free_mnttab(mnttab_hash);
6336                 (void) fclose(tfp);
6337                 (void) unlink(UFS_SIGNATURE_LIST".tmp");
6338                 return (-1);
6339         }

6341         dirp = opendir("/dev/rdsk");
6342         error = errno;
6343         INJECT_ERROR1("OPENDIR_DEV_RDSK", dirp = NULL);
6344         if (dirp == NULL) {
6345                 bam_error(OPENDIR_FAILED, "/dev/rdsk", strerror(error));
6346                 goto fail;
6347         }

6349         while (dp = readdir(dirp)) {
6350                 if (strcmp(dp->d_name, ".") == 0 ||
6351                     strcmp(dp->d_name, "..") == 0)
6352                         continue;

6354                 /*
```



```
6355                  * we only look for the s0 slice. This is guranteed to
6356                  * have 's' at len - 2.
6357                  */
6358                 len = strlen(dp->d_name);
6359                 if (dp->d_name[len - 2 ] != 's' || dp->d_name[len - 1] != '0') {
6360                         BAM_DPRINTF((D_SKIP_SLICE_NOTZERO, fcn, dp->d_name));
6361                         continue;
6362                 }

6364                 ret = process_slice0(dp->d_name, tfp, mnttab_hash, tmpmnt);
6365                 INJECT_ERROR1("PROCESS_S0_FAIL", ret = -1);
6366                 if (ret == -1)
6367                         goto fail;
6368         }

6370         (void) closedir(dirp);
6371         free_mnttab(mnttab_hash);
6372         (void) rmdir(tmpmnt);

6374         ret = fclose(tfp);
6375         error = errno;
6376         INJECT_ERROR1("FCLOSE_SIGNLIST_TMP", ret = EOF);
6377         if (ret == EOF) {
6378                 bam_error(CLOSE_FAIL, UFS_SIGNATURE_LIST".tmp",
6379                     strerror(error));
6380                 (void) unlink(UFS_SIGNATURE_LIST".tmp");
6381                 return (-1);
6382         }

6384         /* We have a list of existing GRUB signatures. Sort it first */
6385         (void) snprintf(cmd, sizeof (cmd),
6386             "/usr/bin/sort -u %s.tmp > %s.sorted",
6387             UFS_SIGNATURE_LIST, UFS_SIGNATURE_LIST);

6389         ret = exec_cmd(cmd, NULL);
6390         INJECT_ERROR1("SORT_SIGN_LIST", ret = 1);
6391         if (ret != 0) {
6392                 bam_error(GRUBSIGN_SORT_FAILED);
6393                 (void) unlink(UFS_SIGNATURE_LIST".sorted");
6394                 (void) unlink(UFS_SIGNATURE_LIST".tmp");
6395                 return (-1);
6396         }

6398         (void) unlink(UFS_SIGNATURE_LIST".tmp");

6400         ret = rename(UFS_SIGNATURE_LIST".sorted", UFS_SIGNATURE_LIST);
6401         error = errno;
6402         INJECT_ERROR1("RENAME_TMP_SIGNLIST", ret = -1);
6403         if (ret == -1) {
6404                 bam_error(RENAME_FAIL, UFS_SIGNATURE_LIST, strerror(error));
6405                 (void) unlink(UFS_SIGNATURE_LIST".sorted");
6406                 return (-1);
6407         }

6409         if (stat(UFS_SIGNATURE_LIST, &sb) == 0 && sb.st_size == 0) {
6410                 BAM_DPRINTF((D_ZERO_LEN_SIGNLIST, fcn, UFS_SIGNATURE_LIST));
6411         }

6413         BAM_DPRINTF((D_RETURN_SUCCESS, fcn));
6414         return (0);

6416 fail:
6417         if (dirp)
6418                 (void) closedir(dirp);
6419         free_mnttab(mnttab_hash);
6420         (void) rmdir(tmpmnt);
```

```
6421         (void) fclose(tfp);
6422         (void) unlink(UFS_SIGNATURE_LIST".tmp");
6423         BAM_DPRINTF((D_RETURN_FAILURE, fcn));
6424         return (-1);
6425 }

6427 static char *
6428 create_ufs_sign(void)
6429 {
6430         struct stat     sb;
6431         int             signnum = -1;
6432         char            tmpsign[MAXNAMELEN + 1];
6433         char            *numstr;
6434         int             i;
6435         FILE            *tfp;
6436         int             ret;
6437         int             error;
6438         const char      *fcn = "create_ufs_sign()";

6440         bam_print(SEARCHING_UFS_SIGN);

6442         ret = FindAllUfsSignatures();
6443         INJECT_ERROR1("FIND_ALL_UFS", ret = -1);
6444         if (ret == -1) {
6445                 bam_error(ERR_FIND_UFS_SIGN);
6446                 return (NULL);
6447         }

6449         /* Make sure the list exists and is owned by root */
6450         INJECT_ERROR1("SIGNLIST_NOT_CREATED",
6451             (void) unlink(UFS_SIGNATURE_LIST));
6452         if (stat(UFS_SIGNATURE_LIST, &sb) == -1 || sb.st_uid != 0) {
6453                 (void) unlink(UFS_SIGNATURE_LIST);
6454                 bam_error(UFS_SIGNATURE_LIST_MISS, UFS_SIGNATURE_LIST);
6455                 return (NULL);
6456         }

6458         if (sb.st_size == 0) {
6459                 bam_print(GRUBSIGN_UFS_NONE);
6460                 i = 0;
6461                 goto found;
6462         }

6464         /* The signature list was sorted when it was created */
6465         tfp = fopen(UFS_SIGNATURE_LIST, "r");
6466         error = errno;
6467         INJECT_ERROR1("FOPEN_SIGN_LIST", tfp = NULL);
6468         if (tfp == NULL) {
6469                 bam_error(UFS_SIGNATURE_LIST_OPENERR,
6470                     UFS_SIGNATURE_LIST, strerror(error));
6471                 (void) unlink(UFS_SIGNATURE_LIST);
6472                 return (NULL);
6473         }

6475         for (i = 0; s_fgets(tmpsign, sizeof (tmpsign), tfp); i++) {

6477                 if (strncmp(tmpsign, GRUBSIGN_UFS_PREFIX,
6478                     strlen(GRUBSIGN_UFS_PREFIX)) != 0) {
6479                         (void) fclose(tfp);
6480                         (void) unlink(UFS_SIGNATURE_LIST);
6481                         bam_error(UFS_BADSIGN, tmpsign);
6482                         return (NULL);
6483                 }
6484                 numstr = tmpsign + strlen(GRUBSIGN_UFS_PREFIX);

6486                 if (numstr[0] == '\0' || !isdigit(numstr[0])) {
```



```
6487                         (void) fclose(tfp);
6488                         (void) unlink(UFS_SIGNATURE_LIST);
6489                         bam_error(UFS_BADSIGN, tmpsign);
6490                         return (NULL);
6491                 }

6493                 signnum = atoi(numstr);
6494                 INJECT_ERROR1("NEGATIVE_SIGN", signnum = -1);
6495                 if (signnum < 0) {
6496                         (void) fclose(tfp);
6497                         (void) unlink(UFS_SIGNATURE_LIST);
6498                         bam_error(UFS_BADSIGN, tmpsign);
6499                         return (NULL);
6500                 }

6502                 if (i != signnum) {
6503                         BAM_DPRINTF((D_FOUND_HOLE_SIGNLIST, fcn, i));
6504                         break;
6505                 }
6506         }

6508         (void) fclose(tfp);

6510 found:
6511         (void) snprintf(tmpsign, sizeof (tmpsign), "rootfs%d", i);

6513         /* add the ufs signature to the /var/run list of signatures */
6514         ret = ufs_add_to_sign_list(tmpsign);
6515         INJECT_ERROR1("UFS_ADD_TO_SIGN_LIST", ret = -1);
6516         if (ret == -1) {
6517                 (void) unlink(UFS_SIGNATURE_LIST);
6518                 bam_error(FAILED_ADD_SIGNLIST, tmpsign);
6519                 return (NULL);
6520         }

6522         BAM_DPRINTF((D_RETURN_SUCCESS, fcn));

6524         return (s_strdup(tmpsign));
6525 }

6527 static char *
6528 get_fstype(char *osroot)
6529 {
6530         FILE            *mntfp;
6531         struct mnttab   mp = {0};
6532         struct mnttab   mpref = {0};
6533         int             error;
6534         int             ret;
6535         const char      *fcn = "get_fstype()";

6537         INJECT_ERROR1("GET_FSTYPE_OSROOT", osroot = NULL);
6538         if (osroot == NULL) {
6539                 bam_error(GET_FSTYPE_ARGS);
6540                 return (NULL);
6541         }

6543         mntfp = fopen(MNTTAB, "r");
6544         error = errno;
6545         INJECT_ERROR1("GET_FSTYPE_FOPEN", mntfp = NULL);
6546         if (mntfp == NULL) {
6547                 bam_error(OPEN_FAIL, MNTTAB, strerror(error));
6548                 return (NULL);
6549         }

6551         if (*osroot == '\0')
6552                 mpref.mnt_mountp = "/";
```

```
6553         else
6554                 mpref.mnt_mountp = osroot;

6556         ret = getmntany(mntfp, &mp, &mpref);
6557         INJECT_ERROR1("GET_FSTYPE_GETMNTANY", ret = 1);
6558         if (ret != 0) {
6559                 bam_error(MNTTAB_MNTPT_NOT_FOUND, osroot, MNTTAB);
6560                 (void) fclose(mntfp);
6561                 return (NULL);
6562         }
6563         (void) fclose(mntfp);

6565         INJECT_ERROR1("GET_FSTYPE_NULL", mp.mnt_fstype = NULL);
6566         if (mp.mnt_fstype == NULL) {
6567                 bam_error(MNTTAB_FSTYPE_NULL, osroot);
6568                 return (NULL);
6569         }

6571         BAM_DPRINTF((D_RETURN_SUCCESS, fcn));

6573         return (s_strdup(mp.mnt_fstype));
6574 }

6576 static char *
6577 create_zfs_sign(char *osdev)
6578 {
6579         char            tmpsign[PATH_MAX];
6580         char            *pool;
6581         const char      *fcn = "create_zfs_sign()";

6583         BAM_DPRINTF((D_FUNC_ENTRY1, fcn, osdev));

6585         /*
6586          * First find the pool name
6587          */
6588         pool = get_pool(osdev);
6589         INJECT_ERROR1("CREATE_ZFS_SIGN_GET_POOL", pool = NULL);
6590         if (pool == NULL) {
6591                 bam_error(GET_POOL_FAILED, osdev);
6592                 return (NULL);
6593         }

6595         (void) snprintf(tmpsign, sizeof (tmpsign), "pool_%s", pool);

6597         BAM_DPRINTF((D_CREATED_ZFS_SIGN, fcn, tmpsign));

6599         free(pool);

6601         BAM_DPRINTF((D_RETURN_SUCCESS, fcn));

6603         return (s_strdup(tmpsign));
6604 }

6606 static char *
6607 create_new_sign(char *osdev, char *fstype)
6608 {
6609         char            *sign;
6610         const char      *fcn = "create_new_sign()";

6612         INJECT_ERROR1("NEW_SIGN_FSTYPE", fstype = "foofs");

6614         if (strcmp(fstype, "zfs") == 0) {
6615                 BAM_DPRINTF((D_CREATE_NEW_ZFS, fcn));
6616                 sign = create_zfs_sign(osdev);
6617         } else if (strcmp(fstype, "ufs") == 0) {
6618                 BAM_DPRINTF((D_CREATE_NEW_UFS, fcn));
```



```
6619                 sign = create_ufs_sign();
6620         } else {
6621                 bam_error(GRUBSIGN_NOTSUP, fstype);
6622                 sign = NULL;
6623         }

6625         BAM_DPRINTF((D_CREATED_NEW_SIGN, fcn, sign ? sign : "<NULL>"));
6626         return (sign);
6627 }

6629 static int
6630 set_backup_common(char *mntpt, char *sign)
6631 {
6632         FILE            *bfp;
6633         char            backup[PATH_MAX];
6634         char            tmpsign[PATH_MAX];
6635         int             error;
6636         char            *bdir;
6637         char            *backup_dup;
6638         struct stat     sb;
6639         int             ret;
6640         const char      *fcn = "set_backup_common()";

6642         (void) snprintf(backup, sizeof (backup), "%s%s",
6643             mntpt, GRUBSIGN_BACKUP);

6645         /* First read the backup */
6646         bfp = fopen(backup, "r");
6647         if (bfp != NULL) {
6648                 while (s_fgets(tmpsign, sizeof (tmpsign), bfp)) {
6649                         if (strcmp(tmpsign, sign) == 0) {
6650                                 BAM_DPRINTF((D_FOUND_IN_BACKUP, fcn, sign));
6651                                 (void) fclose(bfp);
6652                                 return (0);
6653                         }
6654                 }
6655                 (void) fclose(bfp);
6656                 BAM_DPRINTF((D_NOT_FOUND_IN_EXIST_BACKUP, fcn, sign));
6657         } else {
6658                 BAM_DPRINTF((D_BACKUP_NOT_EXIST, fcn, backup));
6659         }

6661         /*
6662          * Didn't find the correct signature. First create
6663          * the directory if necessary.
6664          */

6666         /* dirname() modifies its argument so dup it */
6667         backup_dup = s_strdup(backup);
6668         bdir = dirname(backup_dup);
6669         assert(bdir);

6671         ret = stat(bdir, &sb);
6672         INJECT_ERROR1("SET_BACKUP_STAT", ret = -1);
6673         if (ret == -1) {
6674                 BAM_DPRINTF((D_BACKUP_DIR_NOEXIST, fcn, bdir));
6675                 ret = mkdirp(bdir, DIR_PERMS);
6676                 error = errno;
6677                 INJECT_ERROR1("SET_BACKUP_MKDIRP", ret = -1);
6678                 if (ret == -1) {
6679                         bam_error(GRUBSIGN_BACKUP_MKDIRERR,
6680                             GRUBSIGN_BACKUP, strerror(error));
6681                         free(backup_dup);
6682                         return (-1);
6683                 }
6684         }
```

```
6685         free(backup_dup);
6687         /*
6688          * Open the backup in append mode to add the correct
6689          * signature;
6690          */
6691         bfp = fopen(backup, "a");
6692         error = errno;
6693         INJECT_ERROR1("SET_BACKUP_FOPEN_A", bfp = NULL);
6694         if (bfp == NULL) {
6695                 bam_error(GRUBSIGN_BACKUP_OPENERR,
6696                     GRUBSIGN_BACKUP, strerror(error));
6697                 return (-1);
6698         }
6700         (void) snprintf(tmpsign, sizeof (tmpsign), "%s\n", sign);
6702         ret = fputs(tmpsign, bfp);
6703         error = errno;
6704         INJECT_ERROR1("SET_BACKUP_FPUTS", ret = 0);
6705         if (ret != strlen(tmpsign)) {
6706                 bam_error(GRUBSIGN_BACKUP_WRITEERR,
6707                     GRUBSIGN_BACKUP, strerror(error));
6708                 (void) fclose(bfp);
6709                 return (-1);
6710         }
6712         (void) fclose(bfp);
6714         if (bam_verbose)
6715                 bam_print(GRUBSIGN_BACKUP_UPDATED, GRUBSIGN_BACKUP);
6717         BAM_DPRINTF((D_RETURN_SUCCESS, fcn));
6719         return (0);
6720 }
6722 static int
6723 set_backup_ufs(char *osroot, char *sign)
6724 {
6725         const char      *fcn = "set_backup_ufs()";
6727         BAM_DPRINTF((D_FUNC_ENTRY2, fcn, osroot, sign));
6728         return (set_backup_common(osroot, sign));
6729 }
6731 static int
6732 set_backup_zfs(char *osdev, char *sign)
6733 {
6734         char            *pool;
6735         char            *mntpt;
6736         zfs_mnted_t     mnted;
6737         int             ret;
6738         const char      *fcn = "set_backup_zfs()";
6740         BAM_DPRINTF((D_FUNC_ENTRY2, fcn, osdev, sign));
6742         pool = get_pool(osdev);
6743         INJECT_ERROR1("SET_BACKUP_GET_POOL", pool = NULL);
6744         if (pool == NULL) {
6745                 bam_error(GET_POOL_FAILED, osdev);
6746                 return (-1);
6747         }
6749         mntpt = mount_top_dataset(pool, &mnted);
6750         INJECT_ERROR1("SET_BACKUP_MOUNT_DATASET", mntpt = NULL);
```



```
6751         if (mntpt == NULL) {
6752                 bam_error(FAIL_MNT_TOP_DATASET, pool);
6753                 free(pool);
6754                 return (-1);
6755         }
6757         ret = set_backup_common(mntpt, sign);
6759         (void) umount_top_dataset(pool, mnted, mntpt);
6761         free(pool);
6763         INJECT_ERROR1("SET_BACKUP_ZFS_FAIL", ret = 1);
6764         if (ret == 0) {
6765                 BAM_DPRINTF((D_RETURN_SUCCESS, fcn));
6766         } else {
6767                 BAM_DPRINTF((D_RETURN_FAILURE, fcn));
6768         }
6770         return (ret);
6771 }
6773 static int
6774 set_backup(char *osroot, char *osdev, char *sign, char *fstype)
6775 {
6776         const char      *fcn = "set_backup()";
6777         int             ret;
6779         INJECT_ERROR1("SET_BACKUP_FSTYPE", fstype = "foofs");
6781         if (strcmp(fstype, "ufs") == 0) {
6782                 BAM_DPRINTF((D_SET_BACKUP_UFS, fcn));
6783                 ret = set_backup_ufs(osroot, sign);
6784         } else if (strcmp(fstype, "zfs") == 0) {
6785                 BAM_DPRINTF((D_SET_BACKUP_ZFS, fcn));
6786                 ret = set_backup_zfs(osdev, sign);
6787         } else {
6788                 bam_error(GRUBSIGN_NOTSUP, fstype);
6789                 ret = -1;
6790         }
6792         if (ret == 0) {
6793                 BAM_DPRINTF((D_RETURN_SUCCESS, fcn));
6794         } else {
6795                 BAM_DPRINTF((D_RETURN_FAILURE, fcn));
6796         }
6798         return (ret);
6799 }
6801 static int
6802 set_primary_common(char *mntpt, char *sign)
6803 {
6804         char            signfile[PATH_MAX];
6805         char            signdir[PATH_MAX];
6806         struct stat     sb;
6807         int             fd;
6808         int             error;
6809         int             ret;
6810         const char      *fcn = "set_primary_common()";
6812         (void) snprintf(signfile, sizeof (signfile), "%s/%s/%s",
6813             mntpt, GRUBSIGN_DIR, sign);
6815         if (stat(signfile, &sb) != -1) {
6816                 if (bam_verbose)
```

```
6817                        bam_print(PRIMARY_SIGN_EXISTS, sign);
6818                return (0);
6819        } else {
6820                BAM_DPRINTF((D_PRIMARY_NOT_EXIST, fcn, signfile));
6821        }

6823        (void) snprintf(signdir, sizeof (signdir), "%s/%s",
6824            mntpt, GRUBSIGN_DIR);

6826        if (stat(signdir, &sb) == -1) {
6827                BAM_DPRINTF((D_PRIMARY_DIR_NOEXIST, fcn, signdir));
6828                ret = mkdirp(signdir, DIR_PERMS);
6829                error = errno;
6830                INJECT_ERROR1("SET_PRIMARY_MKDIRP", ret = -1);
6831                if (ret == -1) {
6832                        bam_error(GRUBSIGN_MKDIR_ERR, signdir, strerror(errno));
6833                        return (-1);
6834                }
6835        }

6837        fd = open(signfile, O_RDWR|O_CREAT|O_TRUNC, 0444);
6838        error = errno;
6839        INJECT_ERROR1("PRIMARY_SIGN_CREAT", fd = -1);
6840        if (fd == -1) {
6841                bam_error(GRUBSIGN_PRIMARY_CREATERR, signfile, strerror(error));
6842                return (-1);
6843        }

6845        ret = fsync(fd);
6846        error = errno;
6847        INJECT_ERROR1("PRIMARY_FSYNC", ret = -1);
6848        if (ret != 0) {
6849                bam_error(GRUBSIGN_PRIMARY_SYNCERR, signfile, strerror(error));
6850        }

6852        (void) close(fd);

6854        if (bam_verbose)
6855                bam_print(GRUBSIGN_CREATED_PRIMARY, signfile);

6857        BAM_DPRINTF((D_RETURN_SUCCESS, fcn));

6859        return (0);
6860 }

6862 static int
6863 set_primary_ufs(char *osroot, char *sign)
6864 {
6865        const char      *fcn = "set_primary_ufs()";

6867        BAM_DPRINTF((D_FUNC_ENTRY2, fcn, osroot, sign));
6868        return (set_primary_common(osroot, sign));
6869 }

6871 static int
6872 set_primary_zfs(char *osdev, char *sign)
6873 {
6874        char            *pool;
6875        char            *mntpt;
6876        zfs_mnted_t     mnted;
6877        int             ret;
6878        const char      *fcn = "set_primary_zfs()";

6880        BAM_DPRINTF((D_FUNC_ENTRY2, fcn, osdev, sign));

6882        pool = get_pool(osdev);
```



```
6883        INJECT_ERROR1("SET_PRIMARY_ZFS_GET_POOL", pool = NULL);
6884        if (pool == NULL) {
6885                bam_error(GET_POOL_FAILED, osdev);
6886                return (-1);
6887        }

6889        /* Pool name must exist in the sign */
6890        ret = (strstr(sign, pool) != NULL);
6891        INJECT_ERROR1("SET_PRIMARY_ZFS_POOL_SIGN_INCOMPAT", ret = 0);
6892        if (ret == 0) {
6893                bam_error(POOL_SIGN_INCOMPAT, pool, sign);
6894                free(pool);
6895                return (-1);
6896        }

6898        mntpt = mount_top_dataset(pool, &mnted);
6899        INJECT_ERROR1("SET_PRIMARY_ZFS_MOUNT_DATASET", mntpt = NULL);
6900        if (mntpt == NULL) {
6901                bam_error(FAIL_MNT_TOP_DATASET, pool);
6902                free(pool);
6903                return (-1);
6904        }

6906        ret = set_primary_common(mntpt, sign);

6908        (void) umount_top_dataset(pool, mnted, mntpt);

6910        free(pool);

6912        INJECT_ERROR1("SET_PRIMARY_ZFS_FAIL", ret = 1);
6913        if (ret == 0) {
6914                BAM_DPRINTF((D_RETURN_SUCCESS, fcn));
6915        } else {
6916                BAM_DPRINTF((D_RETURN_FAILURE, fcn));
6917        }

6919        return (ret);
6920 }

6922 static int
6923 set_primary(char *osroot, char *osdev, char *sign, char *fstype)
6924 {
6925        const char      *fcn = "set_primary()";
6926        int             ret;

6928        INJECT_ERROR1("SET_PRIMARY_FSTYPE", fstype = "foofs");
6929        if (strcmp(fstype, "ufs") == 0) {
6930                BAM_DPRINTF((D_SET_PRIMARY_UFS, fcn));
6931                ret = set_primary_ufs(osroot, sign);
6932        } else if (strcmp(fstype, "zfs") == 0) {
6933                BAM_DPRINTF((D_SET_PRIMARY_ZFS, fcn));
6934                ret = set_primary_zfs(osdev, sign);
6935        } else {
6936                bam_error(GRUBSIGN_NOTSUP, fstype);
6937                ret = -1;
6938        }

6940        if (ret == 0) {
6941                BAM_DPRINTF((D_RETURN_SUCCESS, fcn));
6942        } else {
6943                BAM_DPRINTF((D_RETURN_FAILURE, fcn));
6944        }

6946        return (ret);
6947 }
```

```
6949 static int
6950 ufs_add_to_sign_list(char *sign)
6951 {
6952         FILE            *tfp;
6953         char            signline[MAXNAMELEN];
6954         char            cmd[PATH_MAX];
6955         int             ret;
6956         int             error;
6957         const char      *fcn = "ufs_add_to_sign_list()";

6959         INJECT_ERROR1("ADD_TO_SIGN_LIST_NOT_UFS", sign = "pool_rpool5");
6960         if (strncmp(sign, GRUBSIGN_UFS_PREFIX,
6961             strlen(GRUBSIGN_UFS_PREFIX)) != 0) {
6962                 bam_error(INVALID_UFS_SIGN, sign);
6963                 (void) unlink(UFS_SIGNATURE_LIST);
6964                 return (-1);
6965         }

6967         /*
6968          * most failures in this routine are not a fatal error
6969          * We simply unlink the /var/run file and continue
6970          */

6972         ret = rename(UFS_SIGNATURE_LIST, UFS_SIGNATURE_LIST".tmp");
6973         error = errno;
6974         INJECT_ERROR1("ADD_TO_SIGN_LIST_RENAME", ret = -1);
6975         if (ret == -1) {
6976                 bam_error(RENAME_FAIL, UFS_SIGNATURE_LIST".tmp",
6977                     strerror(error));
6978                 (void) unlink(UFS_SIGNATURE_LIST);
6979                 return (0);
6980         }

6982         tfp = fopen(UFS_SIGNATURE_LIST".tmp", "a");
6983         error = errno;
6984         INJECT_ERROR1("ADD_TO_SIGN_LIST_FOPEN", tfp = NULL);
6985         if (tfp == NULL) {
6986                 bam_error(OPEN_FAIL, UFS_SIGNATURE_LIST".tmp", strerror(error));
6987                 (void) unlink(UFS_SIGNATURE_LIST".tmp");
6988                 return (0);
6989         }

6991         (void) snprintf(signline, sizeof (signline), "%s\n", sign);

6993         ret = fputs(signline, tfp);
6994         error = errno;
6995         INJECT_ERROR1("ADD_TO_SIGN_LIST_FPUTS", ret = 0);
6996         if (ret != strlen(signline)) {
6997                 bam_error(SIGN_LIST_FPUTS_ERR, sign, strerror(error));
6998                 (void) fclose(tfp);
6999                 (void) unlink(UFS_SIGNATURE_LIST".tmp");
7000                 return (0);
7001         }

7003         ret = fclose(tfp);
7004         error = errno;
7005         INJECT_ERROR1("ADD_TO_SIGN_LIST_FCLOSE", ret = EOF);
7006         if (ret == EOF) {
7007                 bam_error(CLOSE_FAIL, UFS_SIGNATURE_LIST".tmp",
7008                     strerror(error));
7009                 (void) unlink(UFS_SIGNATURE_LIST".tmp");
7010                 return (0);
7011         }

7013         /* Sort the list again */
7014         (void) snprintf(cmd, sizeof (cmd),
```



```
7015             "/usr/bin/sort -u %s.tmp > %s.sorted",
7016             UFS_SIGNATURE_LIST, UFS_SIGNATURE_LIST);

7018         ret = exec_cmd(cmd, NULL);
7019         INJECT_ERROR1("ADD_TO_SIGN_LIST_SORT", ret = 1);
7020         if (ret != 0) {
7021                 bam_error(GRUBSIGN_SORT_FAILED);
7022                 (void) unlink(UFS_SIGNATURE_LIST".sorted");
7023                 (void) unlink(UFS_SIGNATURE_LIST".tmp");
7024                 return (0);
7025         }

7027         (void) unlink(UFS_SIGNATURE_LIST".tmp");

7029         ret = rename(UFS_SIGNATURE_LIST".sorted", UFS_SIGNATURE_LIST);
7030         error = errno;
7031         INJECT_ERROR1("ADD_TO_SIGN_LIST_RENAME2", ret = -1);
7032         if (ret == -1) {
7033                 bam_error(RENAME_FAIL, UFS_SIGNATURE_LIST, strerror(error));
7034                 (void) unlink(UFS_SIGNATURE_LIST".sorted");
7035                 return (0);
7036         }

7038         BAM_DPRINTF((D_RETURN_SUCCESS, fcn));

7040         return (0);
7041 }

7043 static int
7044 set_signature(char *osroot, char *osdev, char *sign, char *fstype)
7045 {
7046         int             ret;
7047         const char      *fcn = "set_signature()";

7049         BAM_DPRINTF((D_FUNC_ENTRY4, fcn, osroot, osdev, sign, fstype));

7051         ret = set_backup(osroot, osdev, sign, fstype);
7052         INJECT_ERROR1("SET_SIGNATURE_BACKUP", ret = -1);
7053         if (ret == -1) {
7054                 BAM_DPRINTF((D_RETURN_FAILURE, fcn));
7055                 bam_error(SET_BACKUP_FAILED, sign, osroot, osdev);
7056                 return (-1);
7057         }

7059         ret = set_primary(osroot, osdev, sign, fstype);
7060         INJECT_ERROR1("SET_SIGNATURE_PRIMARY", ret = -1);

7062         if (ret == 0) {
7063                 BAM_DPRINTF((D_RETURN_SUCCESS, fcn));
7064         } else {
7065                 BAM_DPRINTF((D_RETURN_FAILURE, fcn));
7066                 bam_error(SET_PRIMARY_FAILED, sign, osroot, osdev);

7068         }
7069         return (ret);
7070 }

7072 char *
7073 get_grubsign(char *osroot, char *osdev)
7074 {
7075         char            *grubsign;      /* (<sign>,#,#) */
7076         char            *slice;
7077         int             fdiskpart;
7078         char            *sign;
7079         char            *fstype;
7080         int             ret;
```

```
7081         const char      *fcn = "get_grubsign()";

7083         BAM_DPRINTF((D_FUNC_ENTRY2, fcn, osroot, osdev));
7084         fstype = get_fstype(osroot);
7085         INJECT_ERROR1("GET_GRUBSIGN_FSTYPE", fstype = NULL);
7086         if (fstype == NULL) {
7087                 bam_error(GET_FSTYPE_FAILED, osroot);
7088                 return (NULL);
7089         }

7091         sign = find_existing_sign(osroot, osdev, fstype);
7092         INJECT_ERROR1("FIND_EXISTING_SIGN", sign = NULL);
7093         if (sign == NULL) {
7094                 BAM_DPRINTF((D_GET_GRUBSIGN_NO_EXISTING, fcn, osroot, osdev));
7095                 sign = create_new_sign(osdev, fstype);
7096                 INJECT_ERROR1("CREATE_NEW_SIGN", sign = NULL);
7097                 if (sign == NULL) {
7098                         bam_error(GRUBSIGN_CREATE_FAIL, osdev);
7099                         free(fstype);
7100                         return (NULL);
7101                 }
7102         }

7104         ret = set_signature(osroot, osdev, sign, fstype);
7105         INJECT_ERROR1("SET_SIGNATURE_FAIL", ret = -1);
7106         if (ret == -1) {
7107                 bam_error(GRUBSIGN_WRITE_FAIL, osdev);
7108                 free(sign);
7109                 free(fstype);
7110                 (void) unlink(UFS_SIGNATURE_LIST);
7111                 return (NULL);
7112         }

7114         free(fstype);

7116         if (bam_verbose)
7117                 bam_print(GRUBSIGN_FOUND_OR_CREATED, sign, osdev);

7119         fdiskpart = get_partition(osdev);
7120         INJECT_ERROR1("GET_GRUBSIGN_FDISK", fdiskpart = -1);
7121         if (fdiskpart == -1) {
7122                 bam_error(FDISKPART_FAIL, osdev);
7123                 free(sign);
7124                 return (NULL);
7125         }

7127         slice = strrchr(osdev, 's');

7129         grubsign = s_calloc(1, MAXNAMELEN + 10);
7130 /*      if (slice) {
4693         if (slice) {
7131                 (void) snprintf(grubsign, MAXNAMELEN + 10, "(%s,%d,%c)",
7132                     sign, fdiskpart, slice[1] + 'a' - '0');
7133         } else
7134                 (void) snprintf(grubsign, MAXNAMELEN + 10, "(%s,%d)",
7135                     sign, fdiskpart);*/
7136         grubsign = strdup(sign);
4698                     sign, fdiskpart);

7138         free(sign);

7140         BAM_DPRINTF((D_GET_GRUBSIGN_SUCCESS, fcn, grubsign));

7142         return (strchr(grubsign,'_') + 1);
4704         return (grubsign);
7143 }
```

*_____unchanged_portion_omitted_*


```
7768 /*
7769  * look for matching bootadm entry with specified parameters
7770  * Here are the rules (based on existing usage):
7771  * - If title is specified, match on title only
7772  * - Else, match on root/findroot, kernel, and module.
7773  *   Note that, if root_opt is non-zero, the absence of
7774  *   root line is considered a match.
7775  */
7776 static entry_t *
7777 find_boot_entry(
7778         menu_t *mp,
7779         char *title,
7780         char *kernel,
7781         char *findroot,
7782         char *root,
7783         char *module,
7784         int root_opt,
7785         int *entry_num)
7786 {
7787         int             i;
7788         line_t          *lp;
7789         entry_t         *ent;
7790         const char      *fcn = "find_boot_entry()";

7792         if (entry_num)
7793                 *entry_num = BAM_ERROR;

7795         /* find matching entry */
7796         for (i = 0, ent = mp->entries; ent; i++, ent = ent->next) {
7797                 lp = ent->start;

7799                 /* first line of entry must be bootadm comment */
7800                 lp = ent->start;
7801                 if (lp->flags != BAM_COMMENT ||
7802                     strcmp(lp->arg, BAM_BOOTADM_HDR) != 0) {
7803                         continue;
7804                 }

7806                 /* advance to title line */
7807                 lp = lp->next;
7808                 if (title) {
7809                         if (lp->flags == BAM_TITLE && lp->arg &&
7810                             strcmp(lp->arg, title) == 0) {
7811                                 BAM_DPRINTF((D_MATCHED_TITLE, fcn, title));
7812                                 break;
7813                         }
7814                         BAM_DPRINTF((D_NOMATCH_TITLE, fcn, title, lp->arg));
7815                         continue;       /* check title only */
7816                 }

7818                 lp = lp->next;  /* advance to root line */
7819                 if (lp == NULL) {
7820                         continue;
7821                 } else if (lp->cmd != NULL &&
7822                     strcmp(lp->cmd, menu_cmds[FINDROOT_CMD]) == 0) {
7823                         INJECT_ERROR1("FIND_BOOT_ENTRY_NULL_FINDROOT",
7824                             findroot = NULL);
7825                         if (findroot == NULL) {
7826                                 BAM_DPRINTF((D_NOMATCH_FINDROOT_NULL,
7827                                     fcn, lp->arg));
7828                                 continue;
7829                         }
7830                         /* findroot command found, try match  */
7831                         if (strncmp(lp->arg, strchr(findroot, '_') + 1, strlen(l
5393                         if (strcmp(lp->arg, findroot) != 0) {
```

```
7832                                BAM_DPRINTF((D_NOMATCH_FINDROOT,
7833                                    fcn, findroot, lp->arg));
7834                                continue;
7835                        }
7836                        BAM_DPRINTF((D_MATCHED_FINDROOT, fcn, findroot));
7837                        lp = lp->next;  /* advance to kernel line */
7838                } else if (lp->cmd != NULL &&
7839                    strcmp(lp->cmd, menu_cmds[ROOT_CMD]) == 0) {
7840                        INJECT_ERROR1("FIND_BOOT_ENTRY_NULL_ROOT", root = NULL);
7841                        if (root == NULL) {
7842                                BAM_DPRINTF((D_NOMATCH_ROOT_NULL,
7843                                    fcn, lp->arg));
7844                                continue;
7845                        }
7846                        /* root cmd found, try match */
7847                        if (strcmp(lp->arg, root) != 0) {
7848                                BAM_DPRINTF((D_NOMATCH_ROOT,
7849                                    fcn, root, lp->arg));
7850                                continue;
7851                        }
7852                        BAM_DPRINTF((D_MATCHED_ROOT, fcn, root));
7853                        lp = lp->next;  /* advance to kernel line */
7854                } else {
7855                        INJECT_ERROR1("FIND_BOOT_ENTRY_ROOT_OPT_NO",
7856                            root_opt = 0);
7857                        INJECT_ERROR1("FIND_BOOT_ENTRY_ROOT_OPT_YES",
7858                            root_opt = 1);
7859                        /* no root command, see if root is optional */
7860                        if (root_opt == 0) {
7861                                BAM_DPRINTF((D_NO_ROOT_OPT, fcn));
7862                                continue;
7863                        }
7864                        BAM_DPRINTF((D_ROOT_OPT, fcn));
7865                }

7867                if (lp == NULL || lp->next == NULL) {
7868                        continue;
7869                }

7871                if (kernel &&
7872                    (!check_cmd(lp->cmd, KERNEL_CMD, lp->arg, kernel))) {
7873                        if (!(ent->flags & BAM_ENTRY_FAILSAFE) ||
7874                            !(ent->flags & BAM_ENTRY_DBOOT) ||
7875                            strcmp(kernel, DIRECT_BOOT_FAILSAFE_LINE) != 0)
7876                                continue;

7878                        ent->flags |= BAM_ENTRY_UPGFSKERNEL;

7880                }
7881                BAM_DPRINTF((D_KERNEL_MATCH, fcn, kernel, lp->arg));

7883                /*
7884                 * Check for matching module entry (failsafe or normal).
7885                 * If it fails to match, we go around the loop again.
7886                 * For xpv entries, there are two module lines, so we
7887                 * do the check twice.
7888                 */
7889                lp = lp->next;  /* advance to options line */
7890 #endif /* ! codereview */
7891                lp = lp->next;  /* advance to module line */
7892                if (check_cmd(lp->cmd, MODULE_CMD, lp->arg, module) ||
7893                    (((lp = lp->next) != NULL) &&
7894                    check_cmd(lp->cmd, MODULE_CMD, lp->arg, module))) {
7895                        /* match found */
7896                        BAM_DPRINTF((D_MODULE_MATCH, fcn, module, lp->arg));
7897                        break;
```



```
7898                }

7900                if (strcmp(module, FAILSAFE_ARCHIVE) == 0 &&
7901                    (strcmp(lp->prev->arg, FAILSAFE_ARCHIVE_32) == 0 ||
7902                    strcmp(lp->prev->arg, FAILSAFE_ARCHIVE_64) == 0)) {
7903                        ent->flags |= BAM_ENTRY_UPGFSMODULE;
7904                        break;
7905                }

7907        }

7909        if (ent && entry_num) {
7910                *entry_num = i;
7911        }

7913        if (ent) {
7914                BAM_DPRINTF((D_RETURN_RET, fcn, i));
7915        } else {
7916                BAM_DPRINTF((D_RETURN_RET, fcn, BAM_ERROR));
7917        }
7918        return (ent);
7919 }

7921 static int
7922 update_boot_entry(menu_t *mp, char *title, char *findroot, char *root,
7923     char *kernel, char *mod_kernel, char *module, int root_opt)
7924 {
7925        int             i;
7926        int             change_kernel = 0;
7927        entry_t         *ent;
7928        line_t          *lp;
7929        line_t          *tlp;
7930        char            linebuf[BAM_MAXLINE];
7931        const char      *fcn = "update_boot_entry()";
7932        char            *label;
7933 #endif /* ! codereview */

7935        /* note: don't match on title, it's updated on upgrade */
7936        ent = find_boot_entry(mp, NULL, kernel, findroot, root, module,
7937            root_opt, &i);
7938        if ((ent == NULL) && (bam_direct == BAM_DIRECT_DBOOT)) {
7939                /*
7940                 * We may be upgrading a kernel from multiboot to
7941                 * directboot.  Look for a multiboot entry. A multiboot
7942                 * entry will not have a findroot line.
7943                 */
7944                ent = find_boot_entry(mp, NULL, "multiboot", NULL, root,
7945                    MULTIBOOT_ARCHIVE, root_opt, &i);
7946                if (ent != NULL) {
7947                        BAM_DPRINTF((D_UPGRADE_FROM_MULTIBOOT, fcn, root));
7948                        change_kernel = 1;
7949                }
7950        } else if (ent) {
7951                BAM_DPRINTF((D_FOUND_FINDROOT, fcn, findroot));
7952        }

7954        if (ent == NULL) {
7955                BAM_DPRINTF((D_ENTRY_NOT_FOUND_CREATING, fcn, findroot));
7956                return (add_boot_entry(mp, title, findroot,
7957                    kernel, mod_kernel, module, NULL));
7958        }

7960        /* replace title of existing entry and update findroot line */
7961        lp = ent->start;
7962        lp = lp->next;  /* title line */
7963        (void) snprintf(linebuf, sizeof (linebuf), "%s%s%s",
```

```
7964             menu_cmds[TITLE_CMD], menu_cmds[SEP_CMD], title);
7965         free(lp->arg);
7966         free(lp->line);
7967         lp->arg = s_strdup(title);
7968         lp->line = s_strdup(linebuf);
7969         BAM_DPRINTF((D_CHANGING_TITLE, fcn, title));

7971         tlp = lp;       /* title line */
7972         lp = lp->next;  /* root line */

7974         /* if no root or findroot command, create a new line_t */
7975         if ((lp->cmd != NULL) && (strcmp(lp->cmd, menu_cmds[ROOT_CMD]) != 0 &&
7976             strcmp(lp->cmd, menu_cmds[FINDROOT_CMD]) != 0)) {
7977                 lp = s_calloc(1, sizeof (line_t));
7978                 bam_add_line(mp, ent, tlp, lp);
7979         } else {
7980                 if (lp->cmd != NULL)
7981                         free(lp->cmd);

7983                 free(lp->sep);
7984                 free(lp->arg);
7985                 free(lp->line);
7986         }

7988         lp->cmd = s_strdup(menu_cmds[FINDROOT_CMD]);
7989         lp->sep = s_strdup(menu_cmds[SEP_CMD]);
7990         label = s_strdup(strchr(findroot, '_') + 1);
7991         *(strchr(label,',')) = 0;
7992         lp->arg = s_strdup(label);
5451         lp->arg = s_strdup(findroot);
7993         (void) snprintf(linebuf, sizeof (linebuf), "%s%s%s",
7994             menu_cmds[FINDROOT_CMD], menu_cmds[SEP_CMD], label);
5453         menu_cmds[FINDROOT_CMD], menu_cmds[SEP_CMD], findroot);
7995         lp->line = s_strdup(linebuf);
7996         free(label);
7997 #endif /* ! codereview */
7998         BAM_DPRINTF((D_ADDING_FINDROOT_LINE, fcn, findroot));

8000         /* kernel line */
8001         lp = lp->next;

8003         if (ent->flags & BAM_ENTRY_UPGFSKERNEL) {
8004                 char            *params = NULL;
8005                 char            *opts = NULL;
8006 #endif /* ! codereview */

8008                 opts = strpbrk(kernel, " \t");
8009                 *opts++ = '\0';
5455                 params = strstr(lp->line, "-s");
5456                 if (params != NULL)
5457                         (void) snprintf(linebuf, sizeof (linebuf), "%s%s%s%s",
5458                             menu_cmds[KERNEL_DOLLAR_CMD], menu_cmds[SEP_CMD],
5459                             kernel, params+2);
5460                 else
8010                 (void) snprintf(linebuf, sizeof (linebuf), "%s%s%s",
8011                     menu_cmds[KERNEL_DOLLAR_CMD], menu_cmds[SEP_CMD],
8012                     kernel);

8014                 if (lp->cmd != NULL)
8015                         free(lp->cmd);

8017                 free(lp->arg);
8018                 free(lp->line);
8019                 lp->cmd = s_strdup(menu_cmds[KERNEL_DOLLAR_CMD]);
8020                 lp->arg = s_strdup(strstr(linebuf, "/"));
8021                 lp->line = s_strdup(linebuf);
```



```
8022                 ent->flags &= ~BAM_ENTRY_UPGFSKERNEL;
8023                 BAM_DPRINTF((D_ADDING_KERNEL_DOLLAR, fcn, lp->prev->cmd));

8025                 lp = lp->next;
8026                 params = strstr(lp->arg, "-s");
8027                 free(lp->arg);
8028                 free(lp->line);
8029                 if (params)
8030                         (void) snprintf(linebuf, sizeof(linebuf), "%s%s%s -s",
8031                                 lp->cmd, menu_cmds[SEP_CMD],opts);
8032                 else
8033                         (void) snprintf(linebuf, sizeof(linebuf), "%s%s%s",
8034                                 lp->cmd, menu_cmds[SEP_CMD],opts);
8035                 lp->line = s_strdup(linebuf);
8036                 lp->arg = s_strdup(strchr(linebuf, '=') + 1);
8037 #endif /* ! codereview */
8038         }

8040         if (change_kernel) {
8041                 char            *opts = NULL;
8042 #endif /* ! codereview */
8043                 /*
8044                  * We're upgrading from multiboot to directboot.
8045                  */
8046                 opts = strpbrk(kernel, " \t");
8047                 *opts++ = '\0';
8048 #endif /* ! codereview */
8049                 if (lp->cmd != NULL &&
8050                     strcmp(lp->cmd, menu_cmds[KERNEL_CMD]) == 0) {
8051                         (void) snprintf(linebuf, sizeof (linebuf), "%s%s%s",
8052                             menu_cmds[KERNEL_DOLLAR_CMD], menu_cmds[SEP_CMD],
8053                             kernel);
8054                         free(lp->cmd);
8055                         free(lp->arg);
8056                         free(lp->line);
8057                         lp->cmd = s_strdup(menu_cmds[KERNEL_DOLLAR_CMD]);
8058                         lp->arg = s_strdup(kernel);
8059                         lp->line = s_strdup(linebuf);
8060                         lp = lp->next;
8061                         BAM_DPRINTF((D_ADDING_KERNEL_DOLLAR, fcn, kernel));
8062                         (void) snprintf(linebuf, sizeof(linebuf), "%s%s%s",
8063                                 lp->cmd, menu_cmds[SEP_CMD],opts);
8064                         free(lp->arg);
8065                         free(lp->line);
8066                         lp->line = s_strdup(linebuf);
8067                         lp->arg = s_strdup(strchr(linebuf, '=') + 1);
8068 #endif /* ! codereview */
8069                 }
8070                 if (lp->cmd != NULL &&
8071                     strcmp(lp->cmd, menu_cmds[MODULE_CMD]) == 0) {
8072                         (void) snprintf(linebuf, sizeof (linebuf), "%s%s%s",
8073                             menu_cmds[MODULE_DOLLAR_CMD], menu_cmds[SEP_CMD],
8074                             module);
8075                         free(lp->cmd);
8076                         free(lp->arg);
8077                         free(lp->line);
8078                         lp->cmd = s_strdup(menu_cmds[MODULE_DOLLAR_CMD]);
8079                         lp->arg = s_strdup(module);
8080                         lp->line = s_strdup(linebuf);
8081                         lp = lp->next;
8082                         BAM_DPRINTF((D_ADDING_MODULE_DOLLAR, fcn, module));
8083                 }
8084         }

8086         /* module line */
8087         lp = lp->next;
```

```
8089         if (ent->flags & BAM_ENTRY_UPGFSMODULE) {
8090                 if (lp->cmd != NULL &&
8091                     strcmp(lp->cmd, menu_cmds[MODULE_CMD]) == 0) {
8092                         (void) snprintf(linebuf, sizeof (linebuf), "%s%s%s",
8093                             menu_cmds[MODULE_DOLLAR_CMD], menu_cmds[SEP_CMD],
8094                             module);
8095                         free(lp->cmd);
8096                         free(lp->arg);
8097                         free(lp->line);
8098                         lp->cmd = s_strdup(menu_cmds[MODULE_DOLLAR_CMD]);
8099                         lp->arg = s_strdup(module);
8100                         lp->line = s_strdup(linebuf);
8101                         lp = lp->next;
8102                         ent->flags &= ~BAM_ENTRY_UPGFSMODULE;
8103                         BAM_DPRINTF((D_ADDING_MODULE_DOLLAR, fcn, module));
8104                 }
8105         }

8107         BAM_DPRINTF((D_RETURN_RET, fcn, i));
8108         return (i);
8109 }

8111 int
8112 root_optional(char *osroot, char *menu_root)
8113 {
8114         char                    *ospecial;
8115         char                    *mspecial;
8116         char                    *slash;
8117         int                     root_opt;
8118         int                     ret1;
8119         int                     ret2;
8120         const char              *fcn = "root_optional()";

8122         BAM_DPRINTF((D_FUNC_ENTRY2, fcn, osroot, menu_root));

8124         /*
8125          * For all filesystems except ZFS, a straight compare of osroot
8126          * and menu_root will tell us if root is optional.
8127          * For ZFS, the situation is complicated by the fact that
8128          * menu_root and osroot are always different
8129          */
8130         ret1 = is_zfs(osroot);
8131         ret2 = is_zfs(menu_root);
8132         INJECT_ERROR1("ROOT_OPT_NOT_ZFS", ret1 = 0);
8133         if (!ret1 || !ret2) {
8134                 BAM_DPRINTF((D_ROOT_OPT_NOT_ZFS, fcn, osroot, menu_root));
8135                 root_opt = (strcmp(osroot, menu_root) == 0);
8136                 goto out;
8137         }

8139         ospecial = get_special(osroot);
8140         INJECT_ERROR1("ROOT_OPTIONAL_OSPECIAL", ospecial = NULL);
8141         if (ospecial == NULL) {
8142                 bam_error(GET_OSROOT_SPECIAL_ERR, osroot);
8143                 return (0);
8144         }
8145         BAM_DPRINTF((D_ROOT_OPTIONAL_OSPECIAL, fcn, ospecial, osroot));

8147         mspecial = get_special(menu_root);
8148         INJECT_ERROR1("ROOT_OPTIONAL_MSPECIAL", mspecial = NULL);
8149         if (mspecial == NULL) {
8150                 bam_error(GET_MENU_ROOT_SPECIAL_ERR, menu_root);
8151                 free(ospecial);
8152                 return (0);
8153         }
```



```
8154         BAM_DPRINTF((D_ROOT_OPTIONAL_MSPECIAL, fcn, mspecial, menu_root));

8156         slash = strchr(ospecial, '/');
8157         if (slash)
8158                 *slash = '\0';
8159         BAM_DPRINTF((D_ROOT_OPTIONAL_FIXED_OSPECIAL, fcn, ospecial, osroot));

8161         root_opt = (strcmp(ospecial, mspecial) == 0);

8163         free(ospecial);
8164         free(mspecial);

8166 out:
8167         INJECT_ERROR1("ROOT_OPTIONAL_NO", root_opt = 0);
8168         INJECT_ERROR1("ROOT_OPTIONAL_YES", root_opt = 1);
8169         if (root_opt) {
8170                 BAM_DPRINTF((D_RETURN_SUCCESS, fcn));
8171         } else {
8172                 BAM_DPRINTF((D_RETURN_FAILURE, fcn));
8173         }

8175         return (root_opt);
8176 }

8178 /*ARGSUSED*/
8179 static error_t
8180 update_entry(menu_t *mp, char *menu_root, char *osdev)
8181 {
8182         int             entry;
8183         char            *grubsign;
8184         char            *grubroot;
8185         char            *title;
8186         char            osroot[PATH_MAX];
8187         char            *failsafe_kernel = NULL;
8188         struct stat     sbuf;
8189         char            failsafe[256];
8190         char            failsafe_64[256];
8191         int             ret;
8192         const char      *fcn = "update_entry()";

8194         assert(mp);
8195         assert(menu_root);
8196         assert(osdev);
8197         assert(bam_root);

8199         BAM_DPRINTF((D_FUNC_ENTRY3, fcn, menu_root, osdev, bam_root));

8201         (void) strlcpy(osroot, bam_root, sizeof (osroot));

8203         title = get_title(osroot);
8204         assert(title);

8206         grubsign = get_grubsign(osroot, osdev);
8207         INJECT_ERROR1("GET_GRUBSIGN_FAIL", grubsign = NULL);
8208         if (grubsign == NULL) {
8209                 bam_error(GET_GRUBSIGN_ERROR, osroot, osdev);
8210                 return (BAM_ERROR);
8211         }

8213         /*
8214          * It is not a fatal error if get_grubroot() fails
8215          * We no longer rely on biosdev to populate the
8216          * menu
8217          */
8218         grubroot = get_grubroot(osroot, osdev, menu_root);
8219         INJECT_ERROR1("GET_GRUBROOT_FAIL", grubroot = NULL);
```

```
8220         if (grubroot) {
8221                 BAM_DPRINTF((D_GET_GRUBROOT_SUCCESS,
8222                     fcn, osroot, osdev, menu_root));
8223         } else {
8224                 BAM_DPRINTF((D_GET_GRUBROOT_FAILURE,
8225                     fcn, osroot, osdev, menu_root));
8226         }

8228         /* add the entry for normal Solaris */
8229         INJECT_ERROR1("UPDATE_ENTRY_MULTIBOOT",
8230             bam_direct = BAM_DIRECT_MULTIBOOT);
8231         if (bam_direct == BAM_DIRECT_DBOOT) {
8232                 entry = update_boot_entry(mp, title, grubsign, grubroot,
8233                     (bam_zfs ? DIRECT_BOOT_KERNEL_ZFS : DIRECT_BOOT_KERNEL),
8234                     NULL, DIRECT_BOOT_ARCHIVE,
8235                     root_optional(osroot, menu_root));
8236                 BAM_DPRINTF((D_UPDATED_BOOT_ENTRY, fcn, bam_zfs, grubsign));
8237                 if ((entry != BAM_ERROR) && (bam_is_hv == BAM_HV_PRESENT)) {
8238                         (void) update_boot_entry(mp, NEW_HV_ENTRY, grubsign,
8239                             grubroot, XEN_MENU, bam_zfs ?
8240                             XEN_KERNEL_MODULE_LINE_ZFS : XEN_KERNEL_MODULE_LINE,
8241                             DIRECT_BOOT_ARCHIVE,
8242                             root_optional(osroot, menu_root));
8243                         BAM_DPRINTF((D_UPDATED_HV_ENTRY,
8244                             fcn, bam_zfs, grubsign));
8245                 }
8246         } else {
8247                 entry = update_boot_entry(mp, title, grubsign, grubroot,
8248                     MULTI_BOOT, NULL, MULTIBOOT_ARCHIVE,
8249                     root_optional(osroot, menu_root));

8251                 BAM_DPRINTF((D_UPDATED_MULTIBOOT_ENTRY, fcn, grubsign));
8252         }

8254         /*
8255          * Add the entry for failsafe archive.  On a bfu'd system, the
8256          * failsafe may be different than the installed kernel.
8257          */
8258         (void) snprintf(failsafe, sizeof (failsafe), "%s%s",
8259             osroot, FAILSAFE_ARCHIVE_32);
8260         (void) snprintf(failsafe_64, sizeof (failsafe_64), "%s%s",
8261             osroot, FAILSAFE_ARCHIVE_64);

8263         /*
8264          * Check if at least one of the two archives exists
8265          * Using $ISADIR as the default line, we have an entry which works
8266          * for both the cases.
8267          */

8269         if (stat(failsafe, &sbuf) == 0 || stat(failsafe_64, &sbuf) == 0) {

8271                 /* Figure out where the kernel line should point */
8272                 (void) snprintf(failsafe, sizeof (failsafe), "%s%s", osroot,
8273                     DIRECT_BOOT_FAILSAFE_32);
8274                 (void) snprintf(failsafe_64, sizeof (failsafe_64), "%s%s",
8275                     osroot, DIRECT_BOOT_FAILSAFE_64);
8276                 if (stat(failsafe, &sbuf) == 0 ||
8277                     stat(failsafe_64, &sbuf) == 0) {
8278                         failsafe_kernel = DIRECT_BOOT_FAILSAFE_LINE;
8279                 } else {
8280                         (void) snprintf(failsafe, sizeof (failsafe), "%s%s",
8281                             osroot, MULTI_BOOT_FAILSAFE);
8282                         if (stat(failsafe, &sbuf) == 0) {
8283                                 failsafe_kernel = MULTI_BOOT_FAILSAFE_LINE;
8284                         }
8285                 }
```



```
8286                 if (failsafe_kernel != NULL) {
8287                         (void) update_boot_entry(mp, FAILSAFE_TITLE, grubsign,
8288                             grubroot, failsafe_kernel, NULL, FAILSAFE_ARCHIVE,
8289                             root_optional(osroot, menu_root));
8290                         BAM_DPRINTF((D_UPDATED_FAILSAFE_ENTRY, fcn,
8291                             failsafe_kernel));
8292                 }
8293         }
8294         free(grubroot);

8296         INJECT_ERROR1("UPDATE_ENTRY_ERROR", entry = BAM_ERROR);
8297         if (entry == BAM_ERROR) {
8298                 bam_error(FAILED_TO_ADD_BOOT_ENTRY, title, grubsign);
8299                 free(grubsign);
8300                 return (BAM_ERROR);
8301         }
8302         free(grubsign);

8304         update_numbering(mp);
8305         ret = set_global(mp, menu_cmds[DEFAULT_CMD], entry);
8306         INJECT_ERROR1("SET_DEFAULT_ERROR", ret = BAM_ERROR);
8307         if (ret == BAM_ERROR) {
8308                 bam_error(SET_DEFAULT_FAILED, entry);
8309         }
8310         BAM_DPRINTF((D_RETURN_SUCCESS, fcn));
8311         return (BAM_WRITE);
8312 }

8314 static void
8315 save_default_entry(menu_t *mp, const char *which)
8316 {
8317         int             lineNum;
8318         int             entryNum;
8319         int             entry = 0;      /* default is 0 */
8320         char            linebuf[BAM_MAXLINE];
8321         line_t          *lp = mp->curdefault;
8322         const char      *fcn = "save_default_entry()";

8324         if (mp->start) {
8325                 lineNum = mp->end->lineNum;
8326                 entryNum = mp->end->entryNum;
8327         } else {
8328                 lineNum = LINE_INIT;
8329                 entryNum = ENTRY_INIT;
8330         }

8332         if (lp)
8333                 entry = s_strtol(lp->arg);

8335         (void) snprintf(linebuf, sizeof (linebuf), "#%s%d", which, entry);
8336         BAM_DPRINTF((D_SAVING_DEFAULT_TO, fcn, linebuf));
8337         line_parser(mp, linebuf, &lineNum, &entryNum);
8338         BAM_DPRINTF((D_SAVED_DEFAULT_TO, fcn, lineNum, entryNum));
8339 }

8341 static void
8342 restore_default_entry(menu_t *mp, const char *which, line_t *lp)
8343 {
8344         int             entry;
8345         char            *str;
8346         const char      *fcn = "restore_default_entry()";

8348         if (lp == NULL) {
8349                 BAM_DPRINTF((D_RESTORE_DEFAULT_NULL, fcn));
8350                 return;         /* nothing to restore */
8351         }
```

```
8353         BAM_DPRINTF((D_RESTORE_DEFAULT_STR, fcn, which));

8355         str = lp->arg + strlen(which);
8356         entry = s_strtol(str);
8357         (void) set_global(mp, menu_cmds[DEFAULT_CMD], entry);

8359         BAM_DPRINTF((D_RESTORED_DEFAULT_TO, fcn, entry));

8361         /* delete saved old default line */
8362         unlink_line(mp, lp);
8363         line_free(lp);
8364 }

8366 /*
8367  * This function is for supporting reboot with args.
8368  * The opt value can be:
8369  * NULL         delete temp entry, if present
8370  * entry=<n>    switches default entry to <n>
8371  * else         treated as boot-args and setup a temperary menu entry
8372  *              and make it the default
8373  * Note that we are always rebooting the current OS instance
8374  * so osroot == / always.
8375  */
8376 #define REBOOT_TITLE    "Solaris_reboot_transient"

8378 /*ARGSUSED*/
8379 static error_t
8380 update_temp(menu_t *mp, char *dummy, char *opt)
8381 {
8382         int             entry;
8383         char            *osdev;
8384         char            *fstype;
8385         char            *sign;
8386         char            *opt_ptr;
8387         char            *path;
8388         char            kernbuf[BUFSIZ];
8389         char            args_buf[BUFSIZ];
8390         char            signbuf[PATH_MAX];
8391         int             ret;
8392         const char      *fcn = "update_temp()";

8394         assert(mp);
8395         assert(dummy == NULL);

8397         /* opt can be NULL */
8398         BAM_DPRINTF((D_FUNC_ENTRY1, fcn, opt ? opt : "<NULL>"));
8399         BAM_DPRINTF((D_BAM_ROOT, fcn, bam_alt_root, bam_root));

8401         if (bam_alt_root || bam_rootlen != 1 ||
8402             strcmp(bam_root, "/") != 0 ||
8403             strcmp(rootbuf, "/") != 0) {
8404                 bam_error(ALT_ROOT_INVALID, bam_root);
8405                 return (BAM_ERROR);
8406         }

8408         /* If no option, delete exiting reboot menu entry */
8409         if (opt == NULL) {
8410                 entry_t         *ent;
8411                 BAM_DPRINTF((D_OPT_NULL, fcn));
8412                 ent = find_boot_entry(mp, REBOOT_TITLE, NULL, NULL,
8413                     NULL, NULL, 0, &entry);
8414                 if (ent == NULL) {      /* not found is ok */
8415                         BAM_DPRINTF((D_TRANSIENT_NOTFOUND, fcn));
8416                         return (BAM_SUCCESS);
8417                 }
```



```
8418                 (void) delete_boot_entry(mp, entry, DBE_PRINTERR);
8419                 restore_default_entry(mp, BAM_OLDDEF, mp->olddefault);
8420                 mp->olddefault = NULL;
8421                 BAM_DPRINTF((D_RESTORED_DEFAULT, fcn));
8422                 BAM_DPRINTF((D_RETURN_SUCCESS, fcn));
8423                 return (BAM_WRITE);
8424         }

8426         /* if entry= is specified, set the default entry */
8427         if (strncmp(opt, "entry=", strlen("entry=")) == 0) {
8428                 int entryNum = s_strtol(opt + strlen("entry="));
8429                 BAM_DPRINTF((D_ENTRY_EQUALS, fcn, opt));
8430                 if (selector(mp, opt, &entry, NULL) == BAM_SUCCESS) {
8431                         /* this is entry=# option */
8432                         ret = set_global(mp, menu_cmds[DEFAULT_CMD], entry);
8433                         BAM_DPRINTF((D_ENTRY_SET_IS, fcn, entry, ret));
8434                         return (ret);
8435                 } else {
8436                         bam_error(SET_DEFAULT_FAILED, entryNum);
8437                         return (BAM_ERROR);
8438                 }
8439         }

8441         /*
8442          * add a new menu entry based on opt and make it the default
8443          */

8445         fstype = get_fstype("/");
8446         INJECT_ERROR1("REBOOT_FSTYPE_NULL", fstype = NULL);
8447         if (fstype == NULL) {
8448                 bam_error(REBOOT_FSTYPE_FAILED);
8449                 return (BAM_ERROR);
8450         }

8452         osdev = get_special("/");
8453         INJECT_ERROR1("REBOOT_SPECIAL_NULL", osdev = NULL);
8454         if (osdev == NULL) {
8455                 free(fstype);
8456                 bam_error(REBOOT_SPECIAL_FAILED);
8457                 return (BAM_ERROR);
8458         }

8460         sign = find_existing_sign("/", osdev, fstype);
8461         INJECT_ERROR1("REBOOT_SIGN_NULL", sign = NULL);
8462         if (sign == NULL) {
8463                 free(fstype);
8464                 free(osdev);
8465                 bam_error(REBOOT_SIGN_FAILED);
8466                 return (BAM_ERROR);
8467         }

8469         free(osdev);
8470         (void) strlcpy(signbuf, sign, sizeof (signbuf));
8471         free(sign);

8473         assert(strchr(signbuf, '(') == NULL && strchr(signbuf, ',') == NULL &&
8474             strchr(signbuf, ')') == NULL);

8476         /*
8477          * There is no alternate root while doing reboot with args
8478          * This version of bootadm is only delivered with a DBOOT
8479          * version of Solaris.
8480          */
8481         INJECT_ERROR1("REBOOT_NOT_DBOOT", bam_direct = BAM_DIRECT_MULTIBOOT);
8482         if (bam_direct != BAM_DIRECT_DBOOT) {
8483                 free(fstype);
```

```
8484                bam_error(REBOOT_DIRECT_FAILED);
8485                return (BAM_ERROR);
8486        }

8488        /* add an entry for Solaris reboot */
8489        if (opt[0] == '-') {
8490                /* It's an option - first see if boot-file is set */
8491                ret = get_kernel(mp, KERNEL_CMD, kernbuf, sizeof (kernbuf));
8492                INJECT_ERROR1("REBOOT_GET_KERNEL", ret = BAM_ERROR);
8493                if (ret != BAM_SUCCESS) {
8494                        free(fstype);
8495                        bam_error(REBOOT_GET_KERNEL_FAILED);
8496                        return (BAM_ERROR);
8497                }
8498                if (kernbuf[0] == '\0')
8499                        (void) strlcpy(kernbuf, DIRECT_BOOT_KERNEL,
8500                            sizeof (kernbuf));
8501                /*
8502                 * If this is a zfs file system and kernbuf does not
8503                 * have "-B $ZFS-BOOTFS" string yet, add it.
8504                 */
8505                if (strcmp(fstype, "zfs") == 0 && !strstr(kernbuf, ZFS_BOOT)) {
8506                        (void) strlcat(kernbuf, " ", sizeof (kernbuf));
8507                        (void) strlcat(kernbuf, ZFS_BOOT, sizeof (kernbuf));
8508                }
8509                (void) strlcat(kernbuf, " ", sizeof (kernbuf));
8510                (void) strlcat(kernbuf, opt, sizeof (kernbuf));
8511                BAM_DPRINTF((D_REBOOT_OPTION, fcn, kernbuf));
8512        } else if (opt[0] == '/') {
8513                /* It's a full path, so write it out. */
8514                (void) strlcpy(kernbuf, opt, sizeof (kernbuf));

8516                /*
8517                 * If someone runs:
8518                 *
8519                 *      # eeprom boot-args='-kd'
8520                 *      # reboot /platform/i86pc/kernel/unix
8521                 *
8522                 * we want to use the boot-args as part of the boot
8523                 * line.  On the other hand, if someone runs:
8524                 *
8525                 *      # reboot "/platform/i86pc/kernel/unix -kd"
8526                 *
8527                 * we don't need to mess with boot-args.  If there's
8528                 * no space in the options string, assume we're in the
8529                 * first case.
8530                 */
8531                if (strchr(opt, ' ') == NULL) {
8532                        ret = get_kernel(mp, ARGS_CMD, args_buf,
8533                            sizeof (args_buf));
8534                        INJECT_ERROR1("REBOOT_GET_ARGS", ret = BAM_ERROR);
8535                        if (ret != BAM_SUCCESS) {
8536                                free(fstype);
8537                                bam_error(REBOOT_GET_ARGS_FAILED);
8538                                return (BAM_ERROR);
8539                        }

8541                        if (args_buf[0] != '\0') {
8542                                (void) strlcat(kernbuf, " ", sizeof (kernbuf));
8543                                (void) strlcat(kernbuf, args_buf,
8544                                    sizeof (kernbuf));
8545                        }
8546                }
8547                BAM_DPRINTF((D_REBOOT_ABSPATH, fcn, kernbuf));
8548        } else {
8549                /*
```



```
8550                 * It may be a partial path, or it may be a partial
8551                 * path followed by options.  Assume that only options
8552                 * follow a space.  If someone sends us a kernel path
8553                 * that includes a space, they deserve to be broken.
8554                 */
8555                opt_ptr = strchr(opt, ' ');
8556                if (opt_ptr != NULL) {
8557                        *opt_ptr = '\0';
8558                }

8560                path = expand_path(opt);
8561                if (path != NULL) {
8562                        (void) strlcpy(kernbuf, path, sizeof (kernbuf));
8563                        free(path);

8565                        /*
8566                         * If there were options given, use those.
8567                         * Otherwise, copy over the default options.
8568                         */
8569                        if (opt_ptr != NULL) {
8570                                /* Restore the space in opt string */
8571                                *opt_ptr = ' ';
8572                                (void) strlcat(kernbuf, opt_ptr,
8573                                    sizeof (kernbuf));
8574                        } else {
8575                                ret = get_kernel(mp, ARGS_CMD, args_buf,
8576                                    sizeof (args_buf));
8577                                INJECT_ERROR1("UPDATE_TEMP_PARTIAL_ARGS",
8578                                    ret = BAM_ERROR);
8579                                if (ret != BAM_SUCCESS) {
8580                                        free(fstype);
8581                                        bam_error(REBOOT_GET_ARGS_FAILED);
8582                                        return (BAM_ERROR);
8583                                }

8585                                if (args_buf[0] != '\0') {
8586                                        (void) strlcat(kernbuf, " ",
8587                                            sizeof (kernbuf));
8588                                        (void) strlcat(kernbuf,
8589                                            args_buf, sizeof (kernbuf));
8590                                }
8591                        }
8592                        BAM_DPRINTF((D_REBOOT_RESOLVED_PARTIAL, fcn, kernbuf));
8593                } else {
8594                        free(fstype);
8595                        bam_error(UNKNOWN_KERNEL, opt);
8596                        bam_print_stderr(UNKNOWN_KERNEL_REBOOT);
8597                        return (BAM_ERROR);
8598                }
8599        }
8600        free(fstype);
8601        entry = add_boot_entry(mp, REBOOT_TITLE, signbuf, kernbuf,
8602            NULL, NULL, NULL);
8603        INJECT_ERROR1("REBOOT_ADD_BOOT_ENTRY", entry = BAM_ERROR);
8604        if (entry == BAM_ERROR) {
8605                bam_error(REBOOT_WITH_ARGS_ADD_ENTRY_FAILED);
8606                return (BAM_ERROR);
8607        }

8609        save_default_entry(mp, BAM_OLDDEF);
8610        ret = set_global(mp, menu_cmds[DEFAULT_CMD], entry);
8611        INJECT_ERROR1("REBOOT_SET_GLOBAL", ret = BAM_ERROR);
8612        if (ret == BAM_ERROR) {
8613                bam_error(REBOOT_SET_DEFAULT_FAILED, entry);
8614        }
8615        BAM_DPRINTF((D_RETURN_SUCCESS, fcn));
```

```
8616         return (BAM_WRITE);
8617 }

8619 error_t
8620 set_global(menu_t *mp, char *globalcmd, int val)
8621 {
8622         line_t          *lp;
8623         line_t          *found;
8624         line_t          *last;
8625         char            *cp;
8626         char            *str;
8627         char            prefix[BAM_MAXLINE];
8628         size_t          len;
8629         const char      *fcn = "set_global()";

8631         assert(mp);
8632         assert(globalcmd);

8634         if (strcmp(globalcmd, menu_cmds[DEFAULT_CMD]) == 0) {
8635                 INJECT_ERROR1("SET_GLOBAL_VAL_NEG", val = -1);
8636                 INJECT_ERROR1("SET_GLOBAL_MENU_EMPTY", mp->end = NULL);
8637                 INJECT_ERROR1("SET_GLOBAL_VAL_TOO_BIG", val = 100);
8638                 if (val < 0 || mp->end == NULL || val > mp->end->entryNum) {
8639                         (void) snprintf(prefix, sizeof (prefix), "%d", val);
8640                         bam_error(INVALID_ENTRY, prefix);
8641                         return (BAM_ERROR);
8642                 }
8643         }

8645         found = last = NULL;
8646         for (lp = mp->start; lp; lp = lp->next) {
8647                 if (lp->flags != BAM_GLOBAL)
8648                         continue;

8650                 last = lp; /* track the last global found */

8652                 INJECT_ERROR1("SET_GLOBAL_NULL_CMD", lp->cmd = NULL);
8653                 if (lp->cmd == NULL) {
8654                         bam_error(NO_CMD, lp->lineNum);
8655                         continue;
8656                 }
8657                 if (strcmp(globalcmd, lp->cmd) != 0)
8658                         continue;

8660                 BAM_DPRINTF((D_FOUND_GLOBAL, fcn, globalcmd));

8662                 if (found) {
8663                         bam_error(DUP_CMD, globalcmd, lp->lineNum, bam_root);
8664                 }
8665                 found = lp;
8666         }

8668         if (found == NULL) {
8669                 lp = s_calloc(1, sizeof (line_t));
8670                 if (last == NULL) {
8671                         lp->next = mp->start;
8672                         mp->start = lp;
8673                         mp->end = (mp->end) ? mp->end : lp;
8674                 } else {
8675                         lp->next = last->next;
8676                         last->next = lp;
8677                         if (lp->next == NULL)
8678                                 mp->end = lp;
8679                 }
8680                 lp->flags = BAM_GLOBAL; /* other fields not needed for writes */
8681                 len = strlen(globalcmd) + strlen(menu_cmds[SEP_CMD]);
```



```
8682                 len += 10;      /* val < 10 digits */
8683                 lp->line = s_calloc(1, len);
8684                 (void) snprintf(lp->line, len, "%s%s%d",
8685                     globalcmd, menu_cmds[SEP_CMD], val);
8686                 BAM_DPRINTF((D_SET_GLOBAL_WROTE_NEW, fcn, lp->line));
8687                 BAM_DPRINTF((D_RETURN_SUCCESS, fcn));
8688                 return (BAM_WRITE);
8689         }

8691         /*
8692          * We are changing an existing entry. Retain any prefix whitespace,
8693          * but overwrite everything else. This preserves tabs added for
8694          * readability.
8695          */
8696         str = found->line;
8697         cp = prefix;
8698         while (*str == ' ' || *str == '\t')
8699                 *(cp++) = *(str++);
8700         *cp = '\0'; /* Terminate prefix */
8701         len = strlen(prefix) + strlen(globalcmd);
8702         len += strlen(menu_cmds[SEP_CMD]) + 10;

8704         free(found->line);
8705         found->line = s_calloc(1, len);
8706         (void) snprintf(found->line, len,
8707             "%s%s%s%d", prefix, globalcmd, menu_cmds[SEP_CMD], val);

8709         BAM_DPRINTF((D_SET_GLOBAL_REPLACED, fcn, found->line));
8710         BAM_DPRINTF((D_RETURN_SUCCESS, fcn));
8711         return (BAM_WRITE); /* need a write to menu */
8712 }

8714 /*
8715  * partial_path may be anything like "kernel/unix" or "kmdb".  Try to
8716  * expand it to a full unix path.  The calling function is expected to
8717  * output a message if an error occurs and NULL is returned.
8718  */
8719 static char *
8720 expand_path(const char *partial_path)
8721 {
8722         int             new_path_len;
8723         char            *new_path;
8724         char            new_path2[PATH_MAX];
8725         struct stat     sb;
8726         const char      *fcn = "expand_path()";

8728         new_path_len = strlen(partial_path) + 64;
8729         new_path = s_calloc(1, new_path_len);

8731         /* First, try the simplest case - something like "kernel/unix" */
8732         (void) snprintf(new_path, new_path_len, "/platform/i86pc/%s",
8733             partial_path);
8734         if (stat(new_path, &sb) == 0) {
8735                 BAM_DPRINTF((D_EXPAND_PATH, fcn, new_path));
8736                 return (new_path);
8737         }

8739         if (strcmp(partial_path, "kmdb") == 0) {
8740                 (void) snprintf(new_path, new_path_len, "%s -k",
8741                     DIRECT_BOOT_KERNEL);
8742                 BAM_DPRINTF((D_EXPAND_PATH, fcn, new_path));
8743                 return (new_path);
8744         }

8746         /*
8747          * We've quickly reached unsupported usage.  Try once more to
```

```
8748          * see if we were just given a glom name.
8749          */
8750         (void) snprintf(new_path, new_path_len, "/platform/i86pc/%s/unix",
8751             partial_path);
8752         (void) snprintf(new_path2, PATH_MAX, "/platform/i86pc/%s/amd64/unix",
8753             partial_path);
8754         if (stat(new_path, &sb) == 0) {
8755                 if (stat(new_path2, &sb) == 0) {
8756                         /*
8757                          * We matched both, so we actually
8758                          * want to write the $ISADIR version.
8759                          */
8760                         (void) snprintf(new_path, new_path_len,
8761                             "/platform/i86pc/kernel/%s/$ISADIR/unix",
8762                             partial_path);
8763                 }
8764                 BAM_DPRINTF((D_EXPAND_PATH, fcn, new_path));
8765                 return (new_path);
8766         }

8768         free(new_path);
8769         BAM_DPRINTF((D_RETURN_FAILURE, fcn));
8770         return (NULL);
8771 }

8773 /*
8774  * The kernel cmd and arg have been changed, so
8775  * check whether the archive line needs to change.
8776  */
8777 static void
8778 set_archive_line(entry_t *entryp, line_t *kernelp)
8779 {
8780         line_t          *lp = entryp->start;
8781         char            *new_archive;
8782         menu_cmd_t      m_cmd;
8783         const char      *fcn = "set_archive_line()";

8785         for (; lp != NULL; lp = lp->next) {
8786                 if (lp->cmd != NULL && strncmp(lp->cmd, menu_cmds[MODULE_CMD],
8787                     sizeof (menu_cmds[MODULE_CMD]) - 1) == 0) {
8788                         break;
8789                 }

8791                 INJECT_ERROR1("SET_ARCHIVE_LINE_END_ENTRY", lp = entryp->end);
8792                 if (lp == entryp->end) {
8793                         BAM_DPRINTF((D_ARCHIVE_LINE_NONE, fcn,
8794                             entryp->entryNum));
8795                         return;
8796                 }
8797         }
8798         INJECT_ERROR1("SET_ARCHIVE_LINE_END_MENU", lp = NULL);
8799         if (lp == NULL) {
8800                 BAM_DPRINTF((D_ARCHIVE_LINE_NONE, fcn, entryp->entryNum));
8801                 return;
8802         }

8804         if (strstr(kernelp->arg, "$ISADIR") != NULL) {
8805                 new_archive = DIRECT_BOOT_ARCHIVE;
8806                 m_cmd = MODULE_DOLLAR_CMD;
8807         } else if (strstr(kernelp->arg, "amd64") != NULL) {
8808                 new_archive = DIRECT_BOOT_ARCHIVE_64;
8809                 m_cmd = MODULE_CMD;
8810         } else {
8811                 new_archive = DIRECT_BOOT_ARCHIVE_32;
8812                 m_cmd = MODULE_CMD;
8813         }
```



```
8815         if (strcmp(lp->arg, new_archive) == 0) {
8816                 BAM_DPRINTF((D_ARCHIVE_LINE_NOCHANGE, fcn, lp->arg));
8817                 return;
8818         }

8820         if (lp->cmd != NULL && strcmp(lp->cmd, menu_cmds[m_cmd]) != 0) {
8821                 free(lp->cmd);
8822                 lp->cmd = s_strdup(menu_cmds[m_cmd]);
8823         }

8825         free(lp->arg);
8826         lp->arg = s_strdup(new_archive);
8827         update_line(lp);
8828         BAM_DPRINTF((D_ARCHIVE_LINE_REPLACED, fcn, lp->line));
8829 }

8831 /*
8832  * Title for an entry to set properties that once went in bootenv.rc.
8833  */
8834 #define BOOTENV_RC_TITLE        "Solaris bootenv rc"

8836 /*
8837  * If path is NULL, return the kernel (optnum == KERNEL_CMD) or arguments
8838  * (optnum == ARGS_CMD) in the argument buf.  If path is a zero-length
8839  * string, reset the value to the default.  If path is a non-zero-length
8840  * string, set the kernel or arguments.
8841  */
8842 static error_t
8843 get_set_kernel(
8844         menu_t *mp,
8845         menu_cmd_t optnum,
8846         char *path,
8847         char *buf,
8848         size_t bufsize)
8849 {
8850         int             entryNum;
8851         int             rv = BAM_SUCCESS;
8852         int             free_new_path = 0;
8853         entry_t         *entryp;
8854         line_t          *ptr;
8855         line_t          *kernelp;
8856         char            *new_arg;
8857         char            *old_args;
8858         char            *space;
8859         char            *new_path;
8860         char            old_space;
8861         size_t          old_kernel_len;
8862         size_t          new_str_len;
8863         char            *fstype;
8864         char            *osdev;
8865         char            *sign;
8866         char            signbuf[PATH_MAX];
8867         int             ret;
8868         const char      *fcn = "get_set_kernel()";

8870         assert(bufsize > 0);

8872         ptr = kernelp = NULL;
8873         new_arg = old_args = space = NULL;
8874         new_path = NULL;
8875         buf[0] = '\0';

8877         INJECT_ERROR1("GET_SET_KERNEL_NOT_DBOOT",
8878             bam_direct = BAM_DIRECT_MULTIBOOT);
8879         if (bam_direct != BAM_DIRECT_DBOOT) {
```

```
8880                bam_error(NOT_DBOOT, optnum == KERNEL_CMD ? "kernel" : "args");
8881                return (BAM_ERROR);
8882        }

8884        /*
8885         * If a user changed the default entry to a non-bootadm controlled
8886         * one, we don't want to mess with it.  Just print an error and
8887         * return.
8888         */
8889        if (mp->curdefault) {
8890                entryNum = s_strtol(mp->curdefault->arg);
8891                for (entryp = mp->entries; entryp; entryp = entryp->next) {
8892                        if (entryp->entryNum == entryNum)
8893                                break;
8894                }
8895                if ((entryp != NULL) &&
8896                    ((entryp->flags & (BAM_ENTRY_BOOTADM|BAM_ENTRY_LU)) == 0)) {
8897                        bam_error(DEFAULT_NOT_BAM);
8898                        return (BAM_ERROR);
8899                }
8900        }

8902        entryp = find_boot_entry(mp, BOOTENV_RC_TITLE, NULL, NULL, NULL, NULL,
8903            0, &entryNum);

8905        if (entryp != NULL) {
8906                for (ptr = entryp->start; ptr && ptr != entryp->end;
8907                    ptr = ptr->next) {
8908                        if (strncmp(ptr->cmd, menu_cmds[KERNEL_CMD],
8909                            sizeof (menu_cmds[KERNEL_CMD]) - 1) == 0) {
8910                                kernelp = ptr;
8911                                break;
8912                        }
8913                }
8914                if (kernelp == NULL) {
8915                        bam_error(NO_KERNEL, entryNum);
8916                        return (BAM_ERROR);
8917                }

8919                old_kernel_len = strcspn(kernelp->arg, " \t");
8920                space = old_args = kernelp->arg + old_kernel_len;
8921                while ((*old_args == ' ') || (*old_args == '\t'))
8922                        old_args++;
8923        }

8925        if (path == NULL) {
8926                if (entryp == NULL) {
8927                        BAM_DPRINTF((D_GET_SET_KERNEL_NO_RC, fcn));
8928                        BAM_DPRINTF((D_RETURN_SUCCESS, fcn));
8929                        return (BAM_SUCCESS);
8930                }
8931                assert(kernelp);
8932                if (optnum == ARGS_CMD) {
8933                        if (old_args[0] != '\0') {
8934                                (void) strlcpy(buf, old_args, bufsize);
8935                                BAM_DPRINTF((D_GET_SET_KERNEL_ARGS, fcn, buf));
8936                        }
8937                } else {
8938                        /*
8939                         * We need to print the kernel, so we just turn the
8940                         * first space into a '\0' and print the beginning.
8941                         * We don't print anything if it's the default kernel.
8942                         */
8943                        old_space = *space;
8944                        *space = '\0';
8945                        if (strcmp(kernelp->arg, DIRECT_BOOT_KERNEL) != 0) {
```



```
8946                                (void) strlcpy(buf, kernelp->arg, bufsize);
8947                                BAM_DPRINTF((D_GET_SET_KERNEL_KERN, fcn, buf));
8948                        }
8949                        *space = old_space;
8950                }
8951                BAM_DPRINTF((D_RETURN_SUCCESS, fcn));
8952                return (BAM_SUCCESS);
8953        }

8955        /*
8956         * First, check if we're resetting an entry to the default.
8957         */
8958        if ((path[0] == '\0') ||
8959            ((optnum == KERNEL_CMD) &&
8960            (strcmp(path, DIRECT_BOOT_KERNEL) == 0))) {
8961                if ((entryp == NULL) || (kernelp == NULL)) {
8962                        /* No previous entry, it's already the default */
8963                        BAM_DPRINTF((D_GET_SET_KERNEL_ALREADY, fcn));
8964                        return (BAM_SUCCESS);
8965                }

8967                /*
8968                 * Check if we can delete the entry.  If we're resetting the
8969                 * kernel command, and the args is already empty, or if we're
8970                 * resetting the args command, and the kernel is already the
8971                 * default, we can restore the old default and delete the entry.
8972                 */
8973                if (((optnum == KERNEL_CMD) &&
8974                    ((old_args == NULL) || (old_args[0] == '\0'))) ||
8975                    ((optnum == ARGS_CMD) &&
8976                    (strncmp(kernelp->arg, DIRECT_BOOT_KERNEL,
8977                    sizeof (DIRECT_BOOT_KERNEL) - 1) == 0))) {
8978                        kernelp = NULL;
8979                        (void) delete_boot_entry(mp, entryNum, DBE_PRINTERR);
8980                        restore_default_entry(mp, BAM_OLD_RC_DEF,
8981                            mp->old_rc_default);
8982                        mp->old_rc_default = NULL;
8983                        rv = BAM_WRITE;
8984                        BAM_DPRINTF((D_GET_SET_KERNEL_RESTORE_DEFAULT, fcn));
8985                        goto done;
8986                }

8988                if (optnum == KERNEL_CMD) {
8989                        /*
8990                         * At this point, we've already checked that old_args
8991                         * and entryp are valid pointers.  The "+ 2" is for
8992                         * a space a the string termination character.
8993                         */
8994                        new_str_len = (sizeof (DIRECT_BOOT_KERNEL) - 1) +
8995                            strlen(old_args) + 2;
8996                        new_arg = s_calloc(1, new_str_len);
8997                        (void) snprintf(new_arg, new_str_len, "%s",
8998                            DIRECT_BOOT_KERNEL);
5475                        (void) snprintf(new_arg, new_str_len, "%s %s",
5476                            DIRECT_BOOT_KERNEL, old_args);
8999                        free(kernelp->arg);
9000                        kernelp->arg = new_arg;

9002                        /*
9003                         * We have changed the kernel line, so we may need
9004                         * to update the archive line as well.
9005                         */
9006                        set_archive_line(entryp, kernelp);
9007                        BAM_DPRINTF((D_GET_SET_KERNEL_RESET_KERNEL_SET_ARG,
9008                            fcn, kernelp->arg));
9009                } else {
```

```
9010                        /*
9011                         * We're resetting the boot args to nothing, so
9012                         * we only need to copy the kernel.  We've already
9013                         * checked that the kernel is not the default.
9014                         */
9015                        new_arg = s_calloc(1, old_kernel_len + 1);
9016                        (void) snprintf(new_arg, old_kernel_len + 1, "%s",
9017                            kernelp->arg);
9018                        free(kernelp->arg);
9019                        kernelp->arg = new_arg;
9020                        BAM_DPRINTF((D_GET_SET_KERNEL_RESET_ARG_SET_KERNEL,
9021                            fcn, kernelp->arg));
9022                }
9023                rv = BAM_WRITE;
9024                goto done;
9025        }

9027        /*
9028         * Expand the kernel file to a full path, if necessary
9029         */
9030        if ((optnum == KERNEL_CMD) && (path[0] != '/')) {
9031                new_path = expand_path(path);
9032                if (new_path == NULL) {
9033                        bam_error(UNKNOWN_KERNEL, path);
9034                        BAM_DPRINTF((D_RETURN_FAILURE, fcn));
9035                        return (BAM_ERROR);
9036                }
9037                free_new_path = 1;
9038        } else {
9039                new_path = path;
9040                free_new_path = 0;
9041        }

9043        /*
9044         * At this point, we know we're setting a new value.  First, take care
9045         * of the case where there was no previous entry.
9046         */
9047        if (entryp == NULL) {

9049                /* Similar to code in update_temp */
9050                fstype = get_fstype("/");
9051                INJECT_ERROR1("GET_SET_KERNEL_FSTYPE", fstype = NULL);
9052                if (fstype == NULL) {
9053                        bam_error(BOOTENV_FSTYPE_FAILED);
9054                        rv = BAM_ERROR;
9055                        goto done;
9056                }

9058                osdev = get_special("/");
9059                INJECT_ERROR1("GET_SET_KERNEL_SPECIAL", osdev = NULL);
9060                if (osdev == NULL) {
9061                        free(fstype);
9062                        bam_error(BOOTENV_SPECIAL_FAILED);
9063                        rv = BAM_ERROR;
9064                        goto done;
9065                }

9067                sign = find_existing_sign("/", osdev, fstype);
9068                INJECT_ERROR1("GET_SET_KERNEL_SIGN", sign = NULL);
9069                if (sign == NULL) {
9070                        free(fstype);
9071                        free(osdev);
9072                        bam_error(BOOTENV_SIGN_FAILED);
9073                        rv = BAM_ERROR;
9074                        goto done;
9075                }
```



```
9077                free(osdev);
9078                (void) strlcpy(signbuf, sign, sizeof (signbuf));
9079                free(sign);
9080                assert(strchr(signbuf, '(') == NULL &&
9081                    strchr(signbuf, ',') == NULL &&
9082                    strchr(signbuf, ')') == NULL);

9084                if (optnum == KERNEL_CMD) {
9085                        if (strcmp(fstype, "zfs") == 0) {
9086                                new_str_len = strlen(new_path) +
9087                                    strlen(ZFS_BOOT) + 8;
9088                                new_arg = s_calloc(1, new_str_len);
9089                                (void) snprintf(new_arg, new_str_len, "%s %s",
9090                                    new_path, ZFS_BOOT);
9091                                BAM_DPRINTF((D_GET_SET_KERNEL_NEW_KERN, fcn,
9092                                    new_arg));
9093                                entryNum = add_boot_entry(mp, BOOTENV_RC_TITLE,
9094                                    signbuf, new_arg, NULL, NULL, NULL);
9095                                free(new_arg);
9096                        } else {
9097                                BAM_DPRINTF((D_GET_SET_KERNEL_NEW_KERN, fcn,
9098                                    new_path));
9099                                entryNum = add_boot_entry(mp, BOOTENV_RC_TITLE,
9100                                    signbuf, new_path, NULL, NULL, NULL);
9101                        }
9102                } else {
9103                        new_str_len = strlen(path) + 8;
9104                        if (strcmp(fstype, "zfs") == 0) {
9105                                new_str_len += strlen(DIRECT_BOOT_KERNEL_ZFS);
9106                                new_arg = s_calloc(1, new_str_len);
9107                                (void) snprintf(new_arg, new_str_len, "%s %s",
9108                                    DIRECT_BOOT_KERNEL_ZFS, path);
9109                        } else {
9110                                new_str_len += strlen(DIRECT_BOOT_KERNEL);
9111                                new_arg = s_calloc(1, new_str_len);
9112                                (void) snprintf(new_arg, new_str_len, "%s %s",
9113                                    DIRECT_BOOT_KERNEL, path);
9114                        }

9116                        BAM_DPRINTF((D_GET_SET_KERNEL_NEW_ARG, fcn, new_arg));
9117                        entryNum = add_boot_entry(mp, BOOTENV_RC_TITLE,
9118                            signbuf, new_arg, NULL, DIRECT_BOOT_ARCHIVE, NULL);
9119                        free(new_arg);
9120                }
9121                free(fstype);
9122                INJECT_ERROR1("GET_SET_KERNEL_ADD_BOOT_ENTRY",
9123                    entryNum = BAM_ERROR);
9124                if (entryNum == BAM_ERROR) {
9125                        bam_error(GET_SET_KERNEL_ADD_BOOT_ENTRY,
9126                            BOOTENV_RC_TITLE);
9127                        rv = BAM_ERROR;
9128                        goto done;
9129                }
9130                save_default_entry(mp, BAM_OLD_RC_DEF);
9131                ret = set_global(mp, menu_cmds[DEFAULT_CMD], entryNum);
9132                INJECT_ERROR1("GET_SET_KERNEL_SET_GLOBAL", ret = BAM_ERROR);
9133                if (ret == BAM_ERROR) {
9134                        bam_error(GET_SET_KERNEL_SET_GLOBAL, entryNum);
9135                }
9136                rv = BAM_WRITE;
9137                goto done;
9138        }

9140        /*
9141         * There was already an bootenv entry which we need to edit.
```

```
9142          */
9143         if (optnum == KERNEL_CMD) {
9144                 new_str_len = strlen(new_path) + strlen(old_args) + 2;
9145                 new_arg = s_calloc(1, new_str_len);
9146                 (void) snprintf(new_arg, new_str_len, "%s %s", new_path,
9147                     old_args);
9148                 free(kernelp->arg);
9149                 kernelp->arg = new_arg;

9151                 /*
9152                  * If we have changed the kernel line, we may need to update
9153                  * the archive line as well.
9154                  */
9155                 set_archive_line(entryp, kernelp);
9156                 BAM_DPRINTF((D_GET_SET_KERNEL_REPLACED_KERNEL_SAME_ARG, fcn,
9157                     kernelp->arg));
9158         } else {
9159                 kernelp = kernelp->next;
9160                 new_str_len = strlen(kernelp->arg) + strlen(path) + 8;
5637                 new_str_len = old_kernel_len + strlen(path) + 8;
9161                 new_arg = s_calloc(1, new_str_len);
9162                 (void) strncpy(new_arg, kernelp->arg, strlen(kernelp->arg));
5639                 (void) strncpy(new_arg, kernelp->arg, old_kernel_len);
9163                 (void) strlcat(new_arg, " ", new_str_len);
9164                 (void) strlcat(new_arg, path, new_str_len);
9165                 free(kernelp->arg);
9166                 kernelp->arg = new_arg;
9167                 BAM_DPRINTF((D_GET_SET_KERNEL_SAME_KERNEL_REPLACED_ARG, fcn,
9168                     kernelp->arg));
9169         }
9170         rv = BAM_WRITE;

9172 done:
9173         if ((rv == BAM_WRITE) && kernelp)
9174                 update_line(kernelp);
9175         if (free_new_path)
9176                 free(new_path);
9177         if (rv == BAM_WRITE) {
9178                 BAM_DPRINTF((D_RETURN_SUCCESS, fcn));
9179         } else {
9180                 BAM_DPRINTF((D_RETURN_FAILURE, fcn));
9181         }
9182         return (rv);
9183 }
```

_____***unchanged_portion_omitted_***

****
9360 Fri Jul 27 03:18:20 2012
new/usr/src/cmd/boot/bootadm/bootadm.h
bootadm.c updated
****
_____***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_***

```
**********************************************************
   23665 Fri Jul 27 03:18:22 2012
new/usr/src/cmd/eeprom/i386/benv.c
eeprom
**********************************************************
______unchanged_portion_omitted_

303 #define GRUBADM_STR     "grubadm: "
303 #define BOOTADM_STR     "bootadm: "

305 /*
306  * grubadm starts all error messages with "grubadm: ".
307  * Add a note so users don't get confused on how they ran grubadm.
306  * bootadm starts all error messages with "bootadm: ".
307  * Add a note so users don't get confused on how they ran bootadm.
308  */
309 static void
310 output_error_msg(const char *msg)
311 {
312         size_t len = sizeof (GRUBADM_STR) - 1;
312         size_t len = sizeof (BOOTADM_STR) - 1;

314         if (strncmp(msg, GRUBADM_STR, len) == 0) {
314         if (strncmp(msg, BOOTADM_STR, len) == 0) {
315                 eeprom_error("error returned from %s\n", msg);
316         } else if (msg[0] != '\0') {
317                 eeprom_error("%s\n", msg);
318         }
319 }

321 static char *
322 get_grubadm_value(char *name, const int quiet)
322 get_bootadm_value(char *name, const int quiet)
323 {
324         char *ptr, *ret_str, *end_ptr, *orig_ptr;
325         char output[BUFSIZ];
326         int is_console, is_kernel = 0;
327         size_t len;

329         is_console = (strcmp(name, "console") == 0);

331         if (strcmp(name, "boot-file") == 0) {
332                 is_kernel = 1;
333                 ptr = "/sbin/grubadm --number -1 --get-kernel 2>&1";
333                 ptr = "/sbin/bootadm set-menu kernel 2>&1";
334         } else if (is_console || (strcmp(name, "boot-args") == 0)) {
335                 ptr = "/sbin/grubadm --number -1 --get-opts 2>&1";
335                 ptr = "/sbin/bootadm set-menu args 2>&1";
336         } else {
337                 eeprom_error("Unknown value in get_grubadm_value: %s\n", name);
337                 eeprom_error("Unknown value in get_bootadm_value: %s\n", name);
338                 return (NULL);
339         }

341         if (exec_cmd(ptr, output, BUFSIZ) != 0) {
342                 if (quiet == 0) {
343                         output_error_msg(output);
344                 }
345                 return (NULL);
346         }

348         if (is_console) {
349                 if ((ptr = strstr(output, "console=")) == NULL) {
350                         return (NULL);
351                 }
352                 ptr += strlen("console=");
```



```
354                 /*
355                  * -B may have comma-separated values.  It may also be
356                  * followed by other flags.
357                  */
358                 len = strcspn(ptr, " \t,");
359                 ret_str = calloc(len + 1, 1);
360                 if (ret_str == NULL) {
361                         eeprom_error(NO_MEM, len + 1);
362                         return (NULL);
363                 }
364                 (void) strncpy(ret_str, ptr, len);
365                 return (ret_str);
366         } else if (is_kernel) {
367                 ret_str = strdup(output);
368                 if (ret_str == NULL)
369                         eeprom_error(NO_MEM, strlen(output) + 1);
370                 return (ret_str);
371         } else {
372                 /* If there's no console setting, we can return */
373                 if ((orig_ptr = strstr(output, "console=")) == NULL) {
374                         return (strdup(output));
375                 }
376                 len = strcspn(orig_ptr, " \t,");
377                 ptr = orig_ptr;
378                 end_ptr = orig_ptr + len + 1;

380                 /* Eat up any white space */
381                 while ((*end_ptr == ' ') || (*end_ptr == '\t'))
382                         end_ptr++;

384                 /*
385                  * If there's data following the console string, copy it.
386                  * If not, cut off the new string.
387                  */
388                 if (*end_ptr == '\0')
389                         *ptr = '\0';

391                 while (*end_ptr != '\0') {
392                         *ptr = *end_ptr;
393                         ptr++;
394                         end_ptr++;
395                 }
396                 *ptr = '\0';
397                 if ((strchr(output, '=') == NULL) &&
398                     (strncmp(output, "-B ", 3) == 0)) {
399                         /*
400                          * Since we removed the console setting, we no
401                          * longer need the initial "-B "
402                          */
403                         orig_ptr = output + 3;
404                 } else {
405                         orig_ptr = output;
406                 }

408                 ret_str = strdup(orig_ptr);
409                 if (ret_str == NULL)
410                         eeprom_error(NO_MEM, strlen(orig_ptr) + 1);
411                 return (ret_str);
412         }
413 }

415 /*
416  * If quiet is 1, print nothing if there is no value.  If quiet is 0, print
417  * a message.  Return 1 if the value is printed, 0 otherwise.
418  */
```

```
419 static int
420 print_grubadm_value(char *name, const int quiet)
420 print_bootadm_value(char *name, const int quiet)
421 {
422         int rv = 0;
423         char *value = get_grubadm_value(name, quiet);
423         char *value = get_bootadm_value(name, quiet);

425         if ((value != NULL) && (value[0] != '\0')) {
426                 (void) printf("%s=%s\n", name, value);
427                 rv = 1;
428         } else if (quiet == 0) {
429                 (void) printf("%s: data not available.\n", name);
430         }

432         if (value != NULL)
433                 free(value);
434         return (rv);
435 }

437 static void
438 print_var(char *name, eplist_t *list)
439 {
440         benv_ent_t *p;
441         char *bootcmd;

443         /*
444          * The console property is kept in both menu.lst and bootenv.rc.  The
445          * menu.lst value takes precedence.
446          */
447         if (strcmp(name, "console") == 0) {
448                 if (print_grubadm_value(name, 1) == 0) {
448                 if (print_bootadm_value(name, 1) == 0) {
449                         if ((p = get_var(name, list)) != NULL) {
450                                 (void) printf("%s=%s\n", name, p->val ?
451                                     p->val : "");
452                         } else {
453                                 (void) printf("%s: data not available.\n",
454                                     name);
455                         }
456                 }
457         } else if (strcmp(name, "bootcmd") == 0) {
458                 bootcmd = getbootcmd();
459                 (void) printf("%s=%s\n", name, bootcmd ? bootcmd : "");
460         } else if ((strcmp(name, "boot-file") == 0) ||
461             (strcmp(name, "boot-args") == 0)) {
462                 (void) print_grubadm_value(name, 0);
462                 (void) print_bootadm_value(name, 0);
463         } else if ((p = get_var(name, list)) == NULL) {
464                 (void) printf("%s: data not available.\n", name);
465         } else {
466                 (void) printf("%s=%s\n", name, p->val ? p->val : "");
467         }
468 }

470 static void
471 print_vars(eplist_t *list)
472 {
473         eplist_t *e;
474         benv_ent_t *p;
475         int console_printed = 0;

477         /*
478          * The console property is kept both in menu.lst and bootenv.rc.
479          * The menu.lst value takes precedence, so try printing that one
480          * first.
```



```
481          */
482         console_printed = print_grubadm_value("console", 1);
482         console_printed = print_bootadm_value("console", 1);

484         for (e = list->next; e != list; e = e->next) {
485                 p = (benv_ent_t *)e->item;
486                 if (p->name != NULL) {
487                         if (((strcmp(p->name, "console") == 0) &&
488                             (console_printed == 1)) ||
489                             ((strcmp(p->name, "boot-file") == 0) ||
490                             (strcmp(p->name, "boot-args") == 0))) {
491                                 /* handle these separately */
492                                 continue;
493                         }
494                         (void) printf("%s=%s\n", p->name, p->val ? p->val : "");
495                 }
496         }
497         (void) print_grubadm_value("boot-file", 1);
498         (void) print_grubadm_value("boot-args", 1);
497         (void) print_bootadm_value("boot-file", 1);
498         (void) print_bootadm_value("boot-args", 1);
499 }
______unchanged_portion_omitted_

525 static void
526 set_grubadm_var(char *name, char *value)
526 set_bootadm_var(char *name, char *value)
527 {
528         char buf[BUFSIZ];
529         char output[BUFSIZ] = "";
530         char *console, *args;
531         int is_console;

533         if (verbose) {
534                 (void) printf("old:");
535                 (void) print_grubadm_value(name, 0);
535                 (void) print_bootadm_value(name, 0);
536         }

538         /*
539          * For security, we single-quote whatever we run on the command line,
540          * and we don't allow single quotes in the string.
541          */
542         if (strchr(value, '\'') != NULL) {
543                 eeprom_error("Single quotes are not allowed "
544                     "in the %s property.\n", name);
545                 return;
546         }

548         is_console = (strcmp(name, "console") == 0);
549         if (strcmp(name, "boot-file") == 0) {
550                 (void) snprintf(buf, BUFSIZ, "/sbin/grubadm --number -1 "
551                     "--set-kernel '%s' 2>&1", value);
550                 (void) snprintf(buf, BUFSIZ, "/sbin/bootadm set-menu "
551                     "kernel='%s' 2>&1", value);
552         } else if (is_console || (strcmp(name, "boot-args") == 0)) {
553                 if (is_console) {
554                         args = get_grubadm_value("boot-args", 1);
554                         args = get_bootadm_value("boot-args", 1);
555                         console = value;
556                 } else {
557                         args = value;
558                         console = get_grubadm_value("console", 1);
558                         console = get_bootadm_value("console", 1);
559                 }
560                 if (((args == NULL) || (args[0] == '\0')) &&
```

```
561                     ((console == NULL) || (console[0] == '\0'))) {
562                         (void) snprintf(buf, BUFSIZ, "/sbin/grubadm --number -1
563                             "--set-opts '-B $ZFS_BOOTFS' 2>&1");
562                         (void) snprintf(buf, BUFSIZ, "/sbin/bootadm set-menu "
563                             "args= 2>&1");
564                 } else if ((args == NULL) || (args[0] == '\0')) {
565                         (void) snprintf(buf, BUFSIZ, "/sbin/grubadm --number -1
566                             "--set-opts '-B console=%s' 2>&1",
565                         (void) snprintf(buf, BUFSIZ, "/sbin/bootadm "
566                             "set-menu args='-B console=%s' 2>&1",
567                             console);
568                 } else if ((console == NULL) || (console[0] == '\0')) {
569                         (void) snprintf(buf, BUFSIZ, "/sbin/grubadm --number -1
570                             "--set-opts '%s' 2>&1", args);
569                         (void) snprintf(buf, BUFSIZ, "/sbin/bootadm "
570                             "set-menu args='%s' 2>&1", args);
571                 } else if (strncmp(args, "-B ", 3) != 0) {
572                         (void) snprintf(buf, BUFSIZ, "/sbin/grubadm --number -1
573                             "--set-opts '-B console=%s %s' 2>&1",
572                         (void) snprintf(buf, BUFSIZ, "/sbin/bootadm "
573                             "set-menu args='-B console=%s %s' 2>&1",
574                             console, args);
575                 } else {
576                         (void) snprintf(buf, BUFSIZ, "/sbin/grubadm --number -1
577                             "--set-opts '-B console=%s,%s' 2>&1",
576                         (void) snprintf(buf, BUFSIZ, "/sbin/bootadm "
577                             "set-menu args='-B console=%s,%s' 2>&1",
578                             console, args + 3);
579                 }
580         } else {
581                 eeprom_error("Unknown value in set_grubadm_value: %s\n", name);
581                 eeprom_error("Unknown value in set_bootadm_value: %s\n", name);
582                 return;
583         }

585         if (exec_cmd(buf, output, BUFSIZ) != 0) {
586                 output_error_msg(output);
587                 return;
588         }

590         if (verbose) {
591                 (void) printf("new:");
592                 (void) print_grubadm_value(name, 0);
592                 (void) print_bootadm_value(name, 0);
593         }
594 }

596 /*
597  * Returns 1 if bootenv.rc was modified, 0 otherwise.
598  */
599 static int
600 set_var(char *name, char *val, eplist_t *list)
601 {
602         benv_ent_t *p;
603         int old_verbose;

605         if (strcmp(name, "bootcmd") == 0)
606                 return (0);

608         if ((strcmp(name, "boot-file") == 0) ||
609             (strcmp(name, "boot-args") == 0)) {
610                 set_grubadm_var(name, val);
610                 set_bootadm_var(name, val);
611                 return (0);
612         }
```



```
614         /*
615          * The console property is kept in two places: menu.lst and bootenv.rc.
616          * Update them both.  We clear verbose to prevent duplicate messages.
617          */
618         if (strcmp(name, "console") == 0) {
619                 old_verbose = verbose;
620                 verbose = 0;
621                 set_grubadm_var(name, val);
621                 set_bootadm_var(name, val);
622                 verbose = old_verbose;
623         }

625         if (verbose) {
626                 (void) printf("old:");
627                 print_var(name, list);
628         }

630         if ((p = get_var(name, list)) != NULL) {
631                 free(p->val);
632                 p->val = strdup(val);
633         } else
634                 add_bent(list, NULL, "setprop", name, val);

636         if (verbose) {
637                 (void) printf("new:");
638                 print_var(name, list);
639         }
640         return (1);
641 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   39500 Fri Jul 27 03:18:25 2012
new/usr/src/cmd/halt/halt.c
halt
**********************************************************
______unchanged_portion_omitted_

 114 static ctidlist_t *ctidlist = NULL;
 115 static ctid_t startdct = -1;

 117 #define FMRI_STARTD_CONTRACT \
 118         "svc:/system/svc/restarter:default/:properties/restarter/contract"

 120 #define BEADM_PROG      "/usr/sbin/beadm"
 121 #define GRUBADM_PROG    "/sbin/grubadm"
 121 #define BOOTADM_PROG    "/sbin/bootadm"
 122 #define ZONEADM_PROG    "/usr/sbin/zoneadm"

 124 /*
 125  * The length of FASTBOOT_MOUNTPOINT must be less than MAXPATHLEN.
 126  */
 127 #define FASTBOOT_MOUNTPOINT     "/tmp/.fastboot.root"

 129 /*
 130  * Fast Reboot related variables
 131  */
 132 static char     fastboot_mounted[MAXPATHLEN];

 134 #if defined(__i386)
 135 static grub_boot_args_t fbarg;
 136 static grub_boot_args_t *fbarg_used;
 137 static int fbarg_entnum = GRUB_ENTRY_DEFAULT;
 138 #endif  /* __i386 */

 140 static int validate_ufs_disk(char *, char *);
 141 static int validate_zfs_pool(char *, char *);

 143 static pid_t
 144 get_initpid()
 145 {
 146         static int init_pid = -1;

 148         if (init_pid == -1) {
 149                 if (zone_getattr(getzoneid(), ZONE_ATTR_INITPID, &init_pid,
 150                     sizeof (init_pid)) != sizeof (init_pid)) {
 151                         assert(errno == ESRCH);
 152                         init_pid = -1;
 153                 }
 154         }
 155         return (init_pid);
 156 }
______unchanged_portion_omitted_

 946 static int
 947 exec_cmd(char * invoke, char * output)
 948 {
 949         FILE * cmd = popen(invoke, "r");
 950         if (! cmd)
 951                 return 0;
 952         fgets(output, 512, cmd);
 953         if (! *output) {
 954                 pclose(cmd);
 955                 return 0;
 956         }
 957         output[strlen(output) - 2] = '\0';
 958         pclose(cmd);
```



```
 959         return 1;
 960 }

 962 #endif /* ! codereview */
 963 /*
 964  * Mount the specified BE.
 965  *
 966  * Upon success returns zero and copies bename string to mountpoint[]
 967  */
 968 static int
 969 fastboot_bename(const char *bename, char *mountpoint, size_t mpsz)
 970 {
 971         int rc;

 973         /*
 974          * Attempt to unmount the BE first in case it's already mounted
 975          * elsewhere.
 976          */
 977         (void) halt_exec(BEADM_PROG, "umount", bename, NULL);

 979         if ((rc = halt_exec(BEADM_PROG, "mount", bename, FASTBOOT_MOUNTPOINT,
 980             NULL)) != 0)
 981                 (void) fprintf(stderr,
 982                     gettext("%s: Unable to mount BE \"%s\" at %s\n"),
 983                     cmdname, bename, FASTBOOT_MOUNTPOINT);
 984         else
 985                 (void) strlcpy(mountpoint, FASTBOOT_MOUNTPOINT, mpsz);

 987         return (rc);
 988 }

 990 /*
 991  * Returns 0 on successful parsing of the arguments;
 992  * returns EINVAL on parsing failures that should abort the reboot attempt;
 993  * returns other error code to fall back to regular reboot.
 994  */
 995 static int
 996 parse_fastboot_args(char *bootargs_buf, size_t buf_size,
 997     int *is_dryrun, const char *bename)
 998 {
 999         char mountpoint[MAXPATHLEN];
1000         char bootargs_saved[BOOTARGS_MAX];
1001         char bootargs_scratch[BOOTARGS_MAX];
1002         char bootfs_arg[BOOTARGS_MAX];
1003         char unixfile[BOOTARGS_MAX];
1004         char *head, *newarg;
1005         int buflen;             /* length of the bootargs_buf */
1006         int mplen;              /* length of the mount point */
1007         int rootlen = 0;        /* length of the root argument */
1008         int unixlen = 0;        /* length of the unix argument */
1009         int off = 0;            /* offset into the new boot argument */
1010         int is_zfs = 0;
1011         int rc = 0;

1013         bzero(mountpoint, sizeof (mountpoint));

1015         /*
1016          * If argc is not 0, buflen is length of the argument being passed in;
1017          * else it is 0 as bootargs_buf has been initialized to all 0's.
1018          */
1019         buflen = strlen(bootargs_buf);

1021         /* Save a copy of the original argument */
1022         bcopy(bootargs_buf, bootargs_saved, buflen);
1023         bzero(&bootargs_saved[buflen], sizeof (bootargs_saved) - buflen);
```

```
1025         /* Save another copy to be used by strtok */
1026         bcopy(bootargs_buf, bootargs_scratch, buflen);
1027         bzero(&bootargs_scratch[buflen], sizeof (bootargs_scratch) - buflen);
1028         head = &bootargs_scratch[0];

1030         /* Get the first argument */
1031         newarg = strtok(bootargs_scratch, " ");

1033         /*
1034          * If this is a dry run request, verify that the drivers can handle
1035          * fast reboot.
1036          */
1037         if (newarg && strncasecmp(newarg, "dryrun", strlen("dryrun")) == 0) {
1038                 *is_dryrun = 1;
1039                 (void) system("/usr/sbin/devfsadm");
1040         }

1042         /*
1043          * Always perform a dry run to identify all the drivers that
1044          * need to implement devo_reset().
1045          */
1046         if (uadmin(A_SHUTDOWN, AD_FASTREBOOT_DRYRUN,
1047             (uintptr_t)bootargs_saved) != 0) {
1048                 (void) fprintf(stderr, gettext("%s: Not all drivers "
1049                     "have implemented quiesce(9E)\n"
1050                     "\tPlease see /var/adm/messages for drivers that haven't\n"
1051                     "\timplemented quiesce(9E).\n"), cmdname);
1052         } else if (*is_dryrun) {
1053                 (void) fprintf(stderr, gettext("%s: All drivers have "
1054                     "implemented quiesce(9E)\n"), cmdname);
1055         }

1057         /* Return if it is a true dry run. */
1058         if (*is_dryrun)
1059                 return (rc);

1061 #if defined(__i386)
1062         /* Read boot args from GRUB menu */
1063         if ((bootargs_buf[0] == 0 || isdigit(bootargs_buf[0])) &&
1064             bename == NULL) {
1065                 /*
1066                  * If no boot arguments are given, or a GRUB menu entry
1067                  * number is provided, process the GRUB menu.
1068                  */
1069                 int entnum;
1070                 if (bootargs_buf[0] == 0)
1071                         entnum = GRUB_ENTRY_DEFAULT;
1072                 else {
1073                         errno = 0;
1074                         entnum = strtoul(bootargs_buf, NULL, 10);
1075                         rc = errno;
1076                 }

1078                 if (rc == 0 && (rc = exec_cmd("/sbin/grubadm --number -1 --get-o
1079                                 fbarg.gba_bootargs)) == 0) {
 946                 if (rc == 0 && (rc = grub_get_boot_args(&fbarg, NULL,
 947                     entnum)) == 0) {
1080                         if (strlcpy(bootargs_buf, fbarg.gba_bootargs,
1081                             buf_size) >= buf_size) {
1082                                 grub_cleanup_boot_args(&fbarg);
1083                                 bcopy(bootargs_saved, bootargs_buf, buf_size);
1084                                 rc = E2BIG;
1085                         }
1086                 }
1087                 /* Failed to read GRUB menu, fall back to normal reboot */
1088                 if (rc != 0) {
```



```
1089                         (void) fprintf(stderr,
1090                             gettext("%s: Failed to process GRUB menu "
1091                             "entry for fast reboot.\n\t%s\n"),
1092                             cmdname, grub_strerror(rc));
1093                         (void) fprintf(stderr,
1094                             gettext("%s: Falling back to regular reboot.\n"),
1095                             cmdname);
1096                         return (-1);
1097                 }
1098                 /* No need to process further */
1099                 fbarg_used = &fbarg;
1100                 fbarg_entnum = entnum;
1101                 return (0);
1102         }
1103 #endif  /* __i386 */

1105         /* Zero out the boot argument buffer as we will reconstruct it */
1106         bzero(bootargs_buf, buf_size);
1107         bzero(bootfs_arg, sizeof (bootfs_arg));
1108         bzero(unixfile, sizeof (unixfile));

1110         if (bename && (rc = fastboot_bename(bename, mountpoint,
1111             sizeof (mountpoint))) != 0)
1112                 return (EINVAL);


1115         /*
1116          * If BE is not specified, look for disk argument to construct
1117          * mountpoint; if BE has been specified, mountpoint has already been
1118          * constructed.
1119          */
1120         if (newarg && newarg[0] != '-' && !bename) {
1121                 int tmprc;

1123                 if ((tmprc = validate_disk(newarg, mountpoint)) == 0) {
1124                         /*
1125                          * The first argument is a valid root argument.
1126                          * Get the next argument.
1127                          */
1128                         newarg = strtok(NULL, " ");
1129                         rootlen = (newarg) ? (newarg - head) : buflen;
1130                         (void) strlcpy(fastboot_mounted, mountpoint,
1131                             sizeof (fastboot_mounted));

1133                 } else if (tmprc == -1) {
1134                         /*
1135                          * Not a disk argument.  Use / as default root.
1136                          */
1137                         bcopy("/", mountpoint, 1);
1138                         bzero(&mountpoint[1], sizeof (mountpoint) - 1);
1139                 } else {
1140                         /*
1141                          * Disk argument, but not valid or not root.
1142                          * Return failure.
1143                          */
1144                         return (EINVAL);
1145                 }
1146         }

1148         /*
1149          * Make mountpoint the first part of unixfile.
1150          * If there is not disk argument, and BE has not been specified,
1151          * mountpoint could be empty.
1152          */
1153         mplen = strlen(mountpoint);
1154         bcopy(mountpoint, unixfile, mplen);
```

```
1156         /*
1157          * Look for unix argument
1158          */
1159         if (newarg && newarg[0] != '-') {
1160                 bcopy(newarg, &unixfile[mplen], strlen(newarg));
1161                 newarg = strtok(NULL, " ");
1162                 rootlen = (newarg) ? (newarg - head) : buflen;
1163         } else if (mplen != 0) {
1164                 /*
1165                  * No unix argument, but mountpoint is not empty, use
1166                  * /platform/i86pc/$ISADIR/kernel/unix as default.
1167                  */
1168                 char isa[20];

1170                 if (sysinfo(SI_ARCHITECTURE_64, isa, sizeof (isa)) != -1)
1171                         (void) snprintf(&unixfile[mplen],
1172                             sizeof (unixfile) - mplen,
1173                             "/platform/i86pc/kernel/%s/unix", isa);
1174                 else if (sysinfo(SI_ARCHITECTURE_32, isa, sizeof (isa)) != -1) {
1175                         (void) snprintf(&unixfile[mplen],
1176                             sizeof (unixfile) - mplen,
1177                             "/platform/i86pc/kernel/unix");
1178                 } else {
1179                         (void) fprintf(stderr,
1180                             gettext("%s: Unknown architecture"), cmdname);
1181                         return (EINVAL);
1182                 }
1183         }

1185         /*
1186          * We now have the complete unix argument.  Verify that it exists and
1187          * is an ELF file.  Split the argument up into mountpoint and unix
1188          * portions again.  This is necessary to handle cases where mountpoint
1189          * is specified on the command line as part of the unix argument,
1190          * such as this:
1191          *      # reboot -f /.alt/platform/i86pc/kernel/amd64/unix
1192          */
1193         unixlen = strlen(unixfile);
1194         if (unixlen > 0) {
1195                 if (validate_unix(unixfile, &mplen, &is_zfs,
1196                     bootfs_arg) != 0) {
1197                         /* Not a valid unix file */
1198                         return (EINVAL);
1199                 } else {
1200                         int space = 0;
1201                         /*
1202                          * Construct boot argument.
1203                          */
1204                         unixlen = strlen(unixfile);

1206                         /*
1207                          * mdep cannot start with space because bootadm
1208                          * creates bogus menu entries if it does.
1209                          */
1210                         if (mplen > 0) {
1211                                 bcopy(unixfile, bootargs_buf, mplen);
1212                                 (void) strcat(bootargs_buf, " ");
1213                                 space = 1;
1214                         }
1215                         bcopy(&unixfile[mplen], &bootargs_buf[mplen + space],
1216                             unixlen - mplen);
1217                         (void) strcat(bootargs_buf, " ");
1218                         off += unixlen + space + 1;
1219                 }
1220         } else {
```



```
1221                 /* Check to see if root is zfs */
1222                 const char      *dp;
1223                 (void) get_zfs_bootfs_arg("/", &dp, &is_zfs, bootfs_arg);
1224         }

1226         if (is_zfs && (buflen != 0 || bename != NULL))  {
1227                 /* LINTED E_SEC_SPRINTF_UNBOUNDED_COPY */
1228                 off += sprintf(bootargs_buf + off, "%s ", bootfs_arg);
1229         }

1231         /*
1232          * Copy the rest of the arguments
1233          */
1234         bcopy(&bootargs_saved[rootlen], &bootargs_buf[off], buflen - rootlen);

1236         return (rc);
1237 }
```

unchanged_portion_omitted_

```
1268 int
1269 main(int argc, char *argv[])
1270 {
1271         char *ttyn = ttyname(STDERR_FILENO);

1273         int qflag = 0, needlog = 1, nosync = 0;
1274         int fast_reboot = 0;
1275         int prom_reboot = 0;
1276         uintptr_t mdep = NULL;
1277         int cmd, fcn, c, aval, r;
1278         const char *usage;
1279         const char *optstring;
1280         zoneid_t zoneid = getzoneid();
1281         int need_check_zones = 0;
1282         char bootargs_buf[BOOTARGS_MAX];
1283         char *bootargs_orig = NULL;
1284         char *bename = NULL;

1286         const char * const resetting = "/etc/svc/volatile/resetting";

1288         (void) setlocale(LC_ALL, "");
1289         (void) textdomain(TEXT_DOMAIN);

1291         cmdname = basename(argv[0]);

1293         if (strcmp(cmdname, "halt") == 0) {
1294                 (void) audit_halt_setup(argc, argv);
1295                 optstring = "dlnqy";
1296                 usage = gettext("usage: %s [ -dlnqy ]\n");
1297                 cmd = A_SHUTDOWN;
1298                 fcn = AD_HALT;
1299         } else if (strcmp(cmdname, "poweroff") == 0) {
1300                 (void) audit_halt_setup(argc, argv);
1301                 optstring = "dlnqy";
1302                 usage = gettext("usage: %s [ -dlnqy ]\n");
1303                 cmd = A_SHUTDOWN;
1304                 fcn = AD_POWEROFF;
1305         } else if (strcmp(cmdname, "reboot") == 0) {
1306                 (void) audit_reboot_setup();
1307 #if defined(__i386)
1308                 optstring = "dlnqpfe:";
1309                 usage = gettext("usage: %s [ -dlnq(p|fe:) ] [ boot args ]\n");
1310 #else
1311                 optstring = "dlnqfp";
1312                 usage = gettext("usage: %s [ -dlnq(p|f) ] [ boot args ]\n");
1313 #endif
1314                 cmd = A_SHUTDOWN;
```

```
1315                 fcn = AD_BOOT;
1316         } else {
1317                 (void) fprintf(stderr,
1318                     gettext("%s: not installed properly\n"), cmdname);
1319                 return (1);
1320         }

1322         while ((c = getopt(argc, argv, optstring)) != EOF) {
1323                 switch (c) {
1324                 case 'd':
1325                         if (zoneid == GLOBAL_ZONEID)
1326                                 cmd = A_DUMP;
1327                         else {
1328                                 (void) fprintf(stderr,
1329                                     gettext("%s: -d only valid from global"
1330                                     " zone\n"), cmdname);
1331                                 return (1);
1332                         }
1333                         break;
1334                 case 'l':
1335                         needlog = 0;
1336                         break;
1337                 case 'n':
1338                         nosync = 1;
1339                         break;
1340                 case 'q':
1341                         qflag = 1;
1342                         break;
1343                 case 'y':
1344                         ttyn = NULL;
1345                         break;
1346                 case 'f':
1347                         fast_reboot = 1;
1348                         break;
1349                 case 'p':
1350                         prom_reboot = 1;
1351                         break;
1352 #if defined(__i386)
1353                 case 'e':
1354                         bename = optarg;
1355                         break;
1356 #endif
1357                 default:
1358                         /*
1359                          * TRANSLATION_NOTE
1360                          * Don't translate the words "halt" or "reboot"
1361                          */
1362                         (void) fprintf(stderr, usage, cmdname);
1363                         return (1);
1364                 }
1365         }

1367         argc -= optind;
1368         argv += optind;

1370         if (argc != 0) {
1371                 if (fcn != AD_BOOT) {
1372                         (void) fprintf(stderr, usage, cmdname);
1373                         return (1);
1374                 }

1376                 /* Gather the arguments into bootargs_buf. */
1377                 if (gather_args(argv, bootargs_buf, sizeof (bootargs_buf)) !=
1378                     0) {
1379                         (void) fprintf(stderr,
1380                             gettext("%s: Boot arguments too long.\n"), cmdname);
```



```
1381                         return (1);
1382                 }

1384                 bootargs_orig = strdup(bootargs_buf);
1385                 mdep = (uintptr_t)bootargs_buf;
1386         } else {
1387                 /*
1388                  * Initialize it to 0 in case of fastboot, the buffer
1389                  * will be used.
1390                  */
1391                 bzero(bootargs_buf, sizeof (bootargs_buf));
1392         }

1394         if (geteuid() != 0) {
1395                 (void) fprintf(stderr,
1396                     gettext("%s: permission denied\n"), cmdname);
1397                 goto fail;
1398         }

1400         if (fast_reboot && prom_reboot) {
1401                 (void) fprintf(stderr,
1402                     gettext("%s: -p and -f are mutually exclusive\n"),
1403                     cmdname);
1404                 return (EINVAL);
1405         }
1406         /*
1407          * Check whether fast reboot is the default operating mode
1408          */
1409         if (fcn == AD_BOOT && !fast_reboot && !prom_reboot &&
1410             zoneid == GLOBAL_ZONEID) {
1411                 fast_reboot = scf_is_fastboot_default();

1413         }

1415         if (bename && !fast_reboot)     {
1416                 (void) fprintf(stderr, gettext("%s: -e only valid with -f\n"),
1417                     cmdname);
1418                 return (EINVAL);
1419         }

1421 #if defined(__sparc)
1422         if (fast_reboot) {
1423                 fast_reboot = 2;        /* need to distinguish each case */
1424         }
1425 #endif

1427         /*
1428          * If fast reboot, do some sanity check on the argument
1429          */
1430         if (fast_reboot == 1) {
1431                 int rc;
1432                 int is_dryrun = 0;

1434                 if (zoneid != GLOBAL_ZONEID)    {
1435                         (void) fprintf(stderr,
1436                             gettext("%s: Fast reboot only valid from global"
1437                             " zone\n"), cmdname);
1438                         return (EINVAL);
1439                 }

1441                 rc = parse_fastboot_args(bootargs_buf, sizeof (bootargs_buf),
1442                     &is_dryrun, bename);

1444                 /*
1445                  * If dry run, or if arguments are invalid, return.
1446                  */
```

```
1447                 if (is_dryrun)
1448                         return (rc);
1449                 else if (rc == EINVAL)
1450                         goto fail;
1451                 else if (rc != 0)
1452                         fast_reboot = 0;

1454                 /*
1455                  * For all the other errors, we continue on in case user
1456                  * user want to force fast reboot, or fall back to regular
1457                  * reboot.
1458                  */
1459                 if (strlen(bootargs_buf) != 0)
1460                         mdep = (uintptr_t)bootargs_buf;
1461         }

1463 #if 0   /* For debugging */
1464         if (mdep != NULL)
1465                 (void) fprintf(stderr, "mdep = %s\n", (char *)mdep);
1466 #endif

1468         if (fcn != AD_BOOT && ttyn != NULL &&
1469             strncmp(ttyn, "/dev/term/", strlen("/dev/term/")) == 0) {
1470                 /*
1471                  * TRANSLATION_NOTE
1472                  * Don't translate ''halt -y''
1473                  */
1474                 (void) fprintf(stderr,
1475                     gettext("%s: dangerous on a dialup;"), cmdname);
1476                 (void) fprintf(stderr,
1477                     gettext("use ''%s -y'' if you are really sure\n"), cmdname);
1478                 goto fail;
1479         }

1481         if (needlog) {
1482                 char *user = getlogin();
1483                 struct passwd *pw;
1484                 char *tty;

1486                 openlog(cmdname, 0, LOG_AUTH);
1487                 if (user == NULL && (pw = getpwuid(getuid())) != NULL)
1488                         user = pw->pw_name;
1489                 if (user == NULL)
1490                         user = "root";

1492                 tty = ttyname(1);

1494                 if (tty == NULL)
1495                         syslog(LOG_CRIT, "initiated by %s", user);
1496                 else
1497                         syslog(LOG_CRIT, "initiated by %s on %s", user, tty);
1498         }

1500         /*
1501          * We must assume success and log it before auditd is terminated.
1502          */
1503         if (fcn == AD_BOOT)
1504                 aval = audit_reboot_success();
1505         else
1506                 aval = audit_halt_success();

1508         if (aval == -1) {
1509                 (void) fprintf(stderr,
1510                     gettext("%s: can't turn off auditd\n"), cmdname);
1511                 if (needlog)
1512                         (void) sleep(5); /* Give syslogd time to record this */
```



```
1513         }

1515         (void) signal(SIGHUP, SIG_IGN); /* for remote connections */

1517         /*
1518          * We start to fork a bunch of zoneadms to halt any active zones.
1519          * This will proceed with halt in parallel until we call
1520          * check_zone_haltedness later on.
1521          */
1522         if (zoneid == GLOBAL_ZONEID && cmd != A_DUMP) {
1523                 need_check_zones = halt_zones();
1524         }

1526 #if defined(__i386)
1527         /* set new default entry in the GRUB entry */
1528         if (fbarg_entnum != GRUB_ENTRY_DEFAULT) {
1529                 char buf[32];
1530                 (void) snprintf(buf, sizeof (buf), "--set-default %u", fbarg_ent
1531                 (void) halt_exec(GRUBADM_PROG, " ", buf, NULL);
1398                 (void) snprintf(buf, sizeof (buf), "default=%u", fbarg_entnum);
1399                 (void) halt_exec(BOOTADM_PROG, "set-menu", buf, NULL);
1532         }
1533 #endif  /* __i386 */

1535         /* if we're dumping, do the archive update here and don't defer it */
1536         if (cmd == A_DUMP && zoneid == GLOBAL_ZONEID && !nosync)
1537                 do_archives_update(fast_reboot);

1539         /*
1540          * If we're not forcing a crash dump, mark the system as quiescing for
1541          * smf(5)'s benefit, and idle the init process.
1542          */
1543         if (cmd != A_DUMP) {
1544                 if (direct_init(PCDSTOP) == -1) {
1545                         /*
1546                          * TRANSLATION_NOTE
1547                          * Don't translate the word "init"
1548                          */
1549                         (void) fprintf(stderr,
1550                             gettext("%s: can't idle init\n"), cmdname);
1551                         goto fail;
1552                 }

1554                 if (creat(resetting, 0755) == -1)
1555                         (void) fprintf(stderr,
1556                             gettext("%s: could not create %s.\n"),
1557                             cmdname, resetting);
1558         }

1560         /*
1561          * Make sure we don't get stopped by a jobcontrol shell
1562          * once we start killing everybody.
1563          */
1564         (void) signal(SIGTSTP, SIG_IGN);
1565         (void) signal(SIGTTIN, SIG_IGN);
1566         (void) signal(SIGTTOU, SIG_IGN);
1567         (void) signal(SIGPIPE, SIG_IGN);
1568         (void) signal(SIGTERM, SIG_IGN);

1570         /*
1571          * Try to stop gdm so X has a chance to return the screen and
1572          * keyboard to a sane state.
1573          */
1574         if (fast_reboot == 1 && stop_gdm() != 0) {
1575                 (void) fprintf(stderr,
1576                     gettext("%s: Falling back to regular reboot.\n"), cmdname);
```

```
1577                 fast_reboot = 0;
1578                 mdep = (uintptr_t)bootargs_orig;
1579         } else if (bootargs_orig) {
1580                 free(bootargs_orig);
1581         }

1583         if (cmd != A_DUMP) {
1584                 /*
1585                  * Stop all restarters so they do not try to restart services
1586                  * that are terminated.
1587                  */
1588                 stop_restarters();

1590                 /*
1591                  * Wait a little while for zones to shutdown.
1592                  */
1593                 if (need_check_zones) {
1594                         check_zones_haltedness();

1596                         (void) fprintf(stderr,
1597                             gettext("%s: Completing system halt.\n"),
1598                             cmdname);
1599                 }
1600         }

1602         /*
1603          * If we're not forcing a crash dump, give everyone 5 seconds to
1604          * handle a SIGTERM and clean up properly.
1605          */
1606         if (cmd != A_DUMP) {
1607                 int     start, end, delta;

1609                 (void) kill(-1, SIGTERM);
1610                 start = time(NULL);

1612                 if (zoneid == GLOBAL_ZONEID && !nosync)
1613                         do_archives_update(fast_reboot);

1615                 end = time(NULL);
1616                 delta = end - start;
1617                 if (delta < 5)
1618                         (void) sleep(5 - delta);
1619         }

1621         (void) signal(SIGINT, SIG_IGN);

1623         if (!qflag && !nosync) {
1624                 struct utmpx wtmpx;

1626                 bzero(&wtmpx, sizeof (struct utmpx));
1627                 (void) strcpy(wtmpx.ut_line, "~");
1628                 (void) time(&wtmpx.ut_tv.tv_sec);

1630                 if (cmd == A_DUMP)
1631                         (void) strcpy(wtmpx.ut_name, "crash dump");
1632                 else
1633                         (void) strcpy(wtmpx.ut_name, "shutdown");

1635                 (void) updwtmpx(WTMPX_FILE, &wtmpx);
1636                 sync();
1637         }

1639         if (cmd == A_DUMP && nosync != 0)
1640                 (void) uadmin(A_DUMP, AD_NOSYNC, NULL);

1642         if (fast_reboot)
```



```
1643                 fcn = AD_FASTREBOOT;

1645         if (uadmin(cmd, fcn, mdep) == -1)
1646                 (void) fprintf(stderr, "%s: uadmin failed: %s\n",
1647                     cmdname, strerror(errno));
1648         else
1649                 (void) fprintf(stderr, "%s: uadmin unexpectedly returned 0\n",
1650                     cmdname);

1652         do {
1653                 r = remove(resetting);
1654         } while (r != 0 && errno == EINTR);

1656         if (r != 0 && errno != ENOENT)
1657                 (void) fprintf(stderr, gettext("%s: could not remove %s.\n"),
1658                     cmdname, resetting);

1660         if (direct_init(PCRUN) == -1) {
1661                 /*
1662                  * TRANSLATION_NOTE
1663                  * Don't translate the word "init"
1664                  */
1665                 (void) fprintf(stderr,
1666                     gettext("%s: can't resume init\n"), cmdname);
1667         }

1669         continue_restarters();

1671         if (get_initpid() != -1)
1672                 /* tell init to restate current level */
1673                 (void) kill(get_initpid(), SIGHUP);

1675 fail:
1676         if (fcn == AD_BOOT)
1677                 (void) audit_reboot_fail();
1678         else
1679                 (void) audit_halt_fail();

1681         if (fast_reboot == 1) {
1682                 if (bename) {
1683                         (void) halt_exec(BEADM_PROG, "umount", bename, NULL);

1685                 } else if (strlen(fastboot_mounted) != 0) {
1686                         (void) umount(fastboot_mounted);
1687 #if defined(__i386)
1688                 } else if (fbarg_used != NULL) {
1689                         grub_cleanup_boot_args(fbarg_used);
1690 #endif  /* __i386 */
1691                 }
1692         }

1694         return (1);
1695 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
  170581 Fri Jul 27 03:18:27 2012
new/usr/src/cmd/svc/startd/graph.c
graph
**********************************************************
______unchanged_portion_omitted_

3577 static int
3578 exec_cmd(char * invoke, char * output)
3579 {
3580         FILE * cmd = popen(invoke, "r");
3581         if (! cmd)
3582                 return 0;
3583         fgets(output, 512, cmd);
3584         if (! *output) {
3585                 pclose(cmd);
3586                 return 0;
3587         }
3588         output[strlen(output) - 2] = '\0';
3589         pclose(cmd);
3590         return 1;
3591 }

3593 #endif /* ! codereview */
3594 static void
3595 do_uadmin(void)
3596 {
3597         const char * const resetting = "/etc/svc/volatile/resetting";
3598         int fd;
3599         struct statvfs vfs;
3600         time_t now;
3601         struct tm nowtm;
3602         char down_buf[256], time_buf[256];
3603         uintptr_t mdep;
3604 #if defined(__i386)
3605         grub_boot_args_t fbarg;
3606 #endif  /* __i386 */

3608         mdep = NULL;
3609         fd = creat(resetting, 0777);
3610         if (fd >= 0)
3611                 startd_close(fd);
3612         else
3613                 uu_warn("Could not create \"%s\"", resetting);

3615         /* Kill dhcpagent if we're not using nfs for root */
3616         if ((statvfs("/", &vfs) == 0) &&
3617             (strncmp(vfs.f_basetype, "nfs", sizeof ("nfs") - 1) != 0))
3618                 fork_with_timeout("/usr/bin/pkill -x -u 0 dhcpagent", 0, 5);

3620         /*
3621          * Call sync(2) now, before we kill off user processes.  This takes
3622          * advantage of the several seconds of pause we have before the
3623          * killalls are done.  Time we can make good use of to get pages
3624          * moving out to disk.
3625          *
3626          * Inside non-global zones, we don't bother, and it's better not to
3627          * anyway, since sync(2) can have system-wide impact.
3628          */
3629         if (getzoneid() == 0)
3630                 sync();

3632         kill_user_procs();

3634         /*
3635          * Note that this must come after the killing of user procs, since
```



```
3636          * killall relies on utmpx, and this command affects the contents of
3637          * said file.
3638          */
3639         if (access("/usr/lib/acct/closewtmp", X_OK) == 0)
3640                 fork_with_timeout("/usr/lib/acct/closewtmp", 0, 5);

3642         /*
3643          * For patches which may be installed as the system is shutting
3644          * down, we need to ensure, one more time, that the boot archive
3645          * really is up to date.
3646          */
3647         if (getzoneid() == 0 && access("/usr/sbin/bootadm", X_OK) == 0)
3648                 fork_with_timeout("/usr/sbin/bootadm -ea update_all", 0, 3600);

3650         /*
3651          * Right now, fast reboot is supported only on i386.
3652          * scf_is_fastboot_default() should take care of it.
3653          * If somehow we got there on unsupported platform -
3654          * print warning and fall back to regular reboot.
3655          */
3656         if (halting == AD_FASTREBOOT) {
3657 #if defined(__i386)
3658                 int rc;

3660                 if ((rc = exec_cmd("/sbin/grubadm --number -1 --get-opts",
3661                         fbarg.gba_bootargs)) == 0) {
3577                 if ((rc = grub_get_boot_args(&fbarg, NULL,
3578                     GRUB_ENTRY_DEFAULT)) == 0) {
3662                         mdep = (uintptr_t)&fbarg.gba_bootargs;
3663                 } else {
3664                         /*
3665                          * Failed to read GRUB menu, fall back to normal reboot
3666                          */
3667                         halting = AD_BOOT;
3668                         uu_warn("Failed to process GRUB menu entry "
3669                             "for fast reboot.\n\t%s\n"
3670                             "Falling back to regular reboot.\n",
3671                             grub_strerror(rc));
3672                 }
3673 #else   /* __i386 */
3674                 halting = AD_BOOT;
3675                 uu_warn("Fast reboot configured, but not supported by "
3676                     "this ISA\n");
3677 #endif  /* __i386 */
3678         }

3680         fork_with_timeout("/sbin/umountall -l", 0, 5);
3681         fork_with_timeout("/sbin/umount /tmp /var/adm /var/run /var "
3682             ">/dev/null 2>&1", 0, 5);

3684         /*
3685          * Try to get to consistency for whatever UFS filesystems are left.
3686          * This is pretty expensive, so we save it for the end in the hopes of
3687          * minimizing what it must do.  The other option would be to start in
3688          * parallel with the killall's, but lockfs tends to throw out much more
3689          * than is needed, and so subsequent commands (like umountall) take a
3690          * long time to get going again.
3691          *
3692          * Inside of zones, we don't bother, since we're not about to terminate
3693          * the whole OS instance.
3694          *
3695          * On systems using only ZFS, this call to lockfs -fa is a no-op.
3696          */
3697         if (getzoneid() == 0) {
3698                 if (access("/usr/sbin/lockfs", X_OK) == 0)
3699                         fork_with_timeout("/usr/sbin/lockfs -fa", 0, 30);
```

```
3701                 sync(); /* once more, with feeling */
3702         }

3704         fork_with_timeout("/sbin/umount /usr >/dev/null 2>&1", 0, 5);

3706         /*
3707          * Construct and emit the last words from userland:
3708          * "<timestamp> The system is down.  Shutdown took <N> seconds."
3709          *
3710          * Normally we'd use syslog, but with /var and other things
3711          * potentially gone, try to minimize the external dependencies.
3712          */
3713         now = time(NULL);
3714         (void) localtime_r(&now, &nowtm);

3716         if (strftime(down_buf, sizeof (down_buf),
3717             "%b %e %T The system is down.", &nowtm) == 0) {
3718                 (void) strlcpy(down_buf, "The system is down.",
3719                     sizeof (down_buf));
3720         }

3722         if (halting_time != 0 && halting_time <= now) {
3723                 (void) snprintf(time_buf, sizeof (time_buf),
3724                     "  Shutdown took %lu seconds.", now - halting_time);
3725         } else {
3726                 time_buf[0] = '\0';
3727         }
3728         (void) printf("%s%s\n", down_buf, time_buf);

3730         (void) uadmin(A_SHUTDOWN, halting, mdep);
3731         uu_warn("uadmin() failed");

3733 #if defined(__i386)
3734         /* uadmin fail, cleanup grub_boot_args */
3735         if (halting == AD_FASTREBOOT)
3736                 grub_cleanup_boot_args(&fbarg);
3737 #endif  /* __i386 */

3739         if (remove(resetting) != 0 && errno != ENOENT)
3740                 uu_warn("Could not remove \"%s\"", resetting);
3741 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
  100804 Fri Jul 27 03:18:30 2012
new/usr/src/lib/libbe/common/be_utils.c
libbe
ba_path->module + lost file
**********************************************************
______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) {
```

```
                        collect_lines = B_FALSE;
                        if ((tok = strtok(NULL, "\n")) == NULL)
                                (void) strlcpy(title, "", sizeof (title));
                        else
                                (void) strlcpy(title, tok, sizeof (title));
                        found_title = B_TRUE;

                        if (num_tmp_lines != 0) {
                                for (i = 0; i < num_tmp_lines; i++) {
                                        free(tmp_entries[i]);
                                        tmp_entries[i] = NULL;
                                }
                                num_tmp_lines = 0;
                        }
                } else if (strcmp(tok, "data_set") == 0) {
                } else if (strcmp(tok, "bootfs") == 0) {
                        char *bootfs = strtok(NULL, BE_WHITE_SPACE);
                        found_title = B_FALSE;
                        if (bootfs == NULL)
                                continue;

                        if (strcmp(bootfs, be_root_ds) == 0) {
                                found_be = B_TRUE;
                                break;
                        }

                        if (be_orig_root_ds != NULL &&
                            strcmp(bootfs, be_orig_root_ds) == 0 &&
                            !found_orig_be) {
                                char str[BUFSIZ];
                                found_orig_be = B_TRUE;
                                num_lines = 0;
                                /*
                                 * Store the new title line
                                 */
                                (void) snprintf(str, BUFSIZ, "entry_name=%s\n",
                                (void) snprintf(str, BUFSIZ, "title %s\n",
                                    description ? description : be_name);
                                entries[num_lines] = strdup(str);
                                num_lines++;
                                /*
                                 * If there are any lines between the title
                                 * and the bootfs line store these. Also
                                 * free the temporary lines.
                                 */
                                for (i = 0; i < num_tmp_lines; i++) {
                                        entries[num_lines] = tmp_entries[i];
                                        tmp_entries[i] = NULL;
                                        num_lines++;
                                }

                                zprop_get_cbdata_t cb = { 0 };
                                zprop_source_t src;
                                char * bc = strchr(be_root_ds, '/');
                                char sguid[] = "guid";

                                *bc = '\0';
                                cb.cb_first = B_TRUE;
                                cb.cb_sources = ZPROP_SRC_ALL;
                                cb.cb_type = ZFS_TYPE_POOL;
                                zprop_get_list(g_zfs, sguid, &cb.cb_proplist, ZF
                                zpool_handle_t * z_hndl = zpool_open(g_zfs, be_r
                                *bc = '/';
                                if (z_hndl) {
                                        uint64_t guid = zpool_get_prop_int(z_hnd
                                        (void) snprintf(str, BUFSIZ, "pool_uuid=
```



```
                                        entries[num_lines] = strdup(str);
                                        num_lines++;
                                        zpool_close(z_hndl);
                                }
num_tmp_lines = 0;
                                num_tmp_lines = 0;
                                /*
                                 * Store the new bootfs line.
                                 */
                                (void) snprintf(str, BUFSIZ, "data_set=%s\n",
                                (void) snprintf(str, BUFSIZ, "bootfs %s\n",
                                    be_root_ds);
                                entries[num_lines] = strdup(str);
                                num_lines++;
                                collect_lines = B_TRUE;
                        }
                } else if (found_orig_be && collect_lines) {
                        /*
                         * get the rest of the lines for the original BE and
                         * store them.
                         */
                        if (strstr(line, BE_GRUB_COMMENT) != NULL ||
                            strstr(line, "BOOTADM") != NULL)
                                continue;
                        if (strcmp(tok, "splashimage") == 0) {
                                entries[num_lines] =
                                    strdup("splashimage "
                                    "/boot/splashimage.xpm\n");
                        } else if ((strcmp(tok, "kernel_path") == 0) ||
                                (strcmp(tok, "module") == 0))
                        {
                                char * path;
                                char * st_path;

                                st_path = strtok(NULL, "\n");
                                path = (char *) malloc(512);
                                strcpy(path, tok);
                                strcat(path, "=");
                                strcat(path, st_path);
                                strcat(path, "\n");
                                entries[num_lines] = path;
#endif /* ! codereview */
                        } else {
                                entries[num_lines] = strdup(temp_line);
                        }
                        num_lines++;
                } else if (found_title && !found_orig_be) {
                        tmp_entries[num_tmp_lines] = strdup(temp_line);
                        num_tmp_lines++;
                }
        }

        (void) fclose(menu_fp);

        if (found_be) {
                /*
                 * If an entry for this BE was already in the menu, then if
                 * that entry's title matches what we would have put in
                 * return success.  Otherwise return failure.
                 */
                char *new_title = description ? description : be_name;

                if (strcmp(title, new_title) == 0) {
                        ret = BE_SUCCESS;
                        goto cleanup;
                } 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",
539                 (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_
```

```
**********************************************************
    7575 Fri Jul 27 03:18:33 2012
new/usr/src/lib/libbe/common/libbe_priv.h
ba_path->module + lost file
**********************************************************
```

```
/*
 * 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) 2008, 2010, Oracle and/or its affiliates. All rights reserved.
 */

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

#ifndef _LIBBE_PRIV_H
#define _LIBBE_PRIV_H

#include <libnvpair.h>
#include <libzfs.h>
#include <instzones_api.h>

#ifdef __cplusplus
extern "C" {
#endif

#define ARCH_LENGTH             MAXNAMELEN
#define BE_AUTO_NAME_MAX_TRY    3
#define BE_AUTO_NAME_DELIM      '-'
#define BE_DEFAULTS             "/etc/default/be"
#define BE_DFLT_BENAME_STARTS   "BENAME_STARTS_WITH="
#define BE_CONTAINER_DS_NAME    "ROOT"
#define BE_DEFAULT_CONSOLE      "text"
#define BE_POLICY_PROPERTY      "org.opensolaris.libbe:policy"
#define BE_UUID_PROPERTY        "org.opensolaris.libbe:uuid"
#define BE_PLCY_STATIC          "static"
#define BE_PLCY_VOLATILE        "volatile"
#define BE_GRUB_MENU            "/boot/illumos.cfg"
#define BE_GRUB_MENU            "/boot/grub/menu.lst"
#define BE_SPARC_MENU           "/boot/menu.lst"
#define BE_GRUB_COMMENT         "#============ End of LIBBE entry ============"
#define BE_GRUB_SPLASH          "splashimage /boot/solaris.xpm"
#define BE_GRUB_FOREGROUND      "foreground 343434"
#define BE_GRUB_BACKGROUND      "background F7FBFF"
#define BE_GRUB_DEFAULT         "default 0"
#define BE_WHITE_SPACE          " \t\r\n"
#define BE_CAP_FILE             "/boot/grub/capability"
```



```
#define BE_INSTALL_GRUB         "/sbin/installgrub"
#define BE_STAGE_1              "/boot/grub/stage1"
#define BE_STAGE_2              "/boot/grub/stage2"
#define ZFS_CLOSE(_zhp) \
        if (_zhp) { \
                zfs_close(_zhp); \
                _zhp = NULL; \
        }

#define BE_ZONE_PARENTBE_PROPERTY       "org.opensolaris.libbe:parentbe"
#define BE_ZONE_ACTIVE_PROPERTY         "org.opensolaris.libbe:active"
#define BE_ZONE_SUPPORTED_BRANDS        "ipkg labeled"
#define BE_ZONE_SUPPORTED_BRANDS_DELIM  " "

/* Maximum length for the BE name. */
#define BE_NAME_MAX_LEN         64

#define MAX(a, b) ((a) > (b) ? (a) : (b))
#define MIN(a, b) ((a) < (b) ? (a) : (b))

typedef struct be_transaction_data {
        char            *obe_name;      /* Original BE name */
        char            *obe_root_ds;   /* Original BE root dataset */
        char            *obe_zpool;     /* Original BE pool */
        char            *obe_snap_name; /* Original BE snapshot name */
        char            *obe_altroot;   /* Original BE altroot */
        char            *nbe_name;      /* New BE name */
        char            *nbe_root_ds;   /* New BE root dataset */
        char            *nbe_zpool;     /* New BE pool */
        char            *nbe_desc;      /* New BE description */
        nvlist_t        *nbe_zfs_props; /* New BE dataset properties */
        char            *policy;        /* BE policy type */
} be_transaction_data_t;
______unchanged_portion_omitted_
```

```
**********************************************************
    7187 Fri Jul 27 03:18:36 2012
new/usr/src/lib/libc/amd64/sys/uadmin.c
uadmin
**********************************************************
______unchanged_portion_omitted_

 164 static char quote[] = "\'";

 166 int
 167 uadmin(int cmd, int fcn, uintptr_t mdep)
 168 {
 169         extern int __uadmin(int cmd, int fcn, uintptr_t mdep);
 170         char *bargs, cmdbuf[256];
 171         struct stat sbuf;
 172         char *altroot;

 174         bargs = (char *)mdep;

 176         if (geteuid() == 0 && getzoneid() == GLOBAL_ZONEID &&
 177             (cmd == A_SHUTDOWN || cmd == A_REBOOT)) {
 178                 int off = 0;

 180                 switch (fcn) {
 181                 case AD_IBOOT:
 182                 case AD_SBOOT:
 183                 case AD_SIBOOT:
 184                         /*
 185                          * These functions fabricate appropriate bootargs.
 186                          * If bootargs are passed in, map these functions
 187                          * to AD_BOOT.
 188                          */
 189                         if (bargs == 0) {
 190                                 switch (fcn) {
 191                                 case AD_IBOOT:
 192                                         bargs = "-a";
 193                                         break;
 194                                 case AD_SBOOT:
 195                                         bargs = "-s";
 196                                         break;
 197                                 case AD_SIBOOT:
 198                                         bargs = "-sa";
 199                                         break;
 200                                 }
 201                         }
 202                         /*FALLTHROUGH*/
 203                 case AD_BOOT:
 204                 case AD_FASTREBOOT:
 205                         if (bargs == 0)
 206                                 break;  /* no args */
 207                         if (legal_arg(bargs) < 0)
 208                                 break;  /* bad args */

 210                         /* avoid cancellation in system() */
 211                         (void) pthread_setcancelstate(PTHREAD_CANCEL_DISABLE,
 212                             NULL);

 214                         /* check for /stubboot */
 215                         if (stat("/stubboot/boot/grub/menu.lst", &sbuf) == 0) {
 216                                 altroot = "-R /stubboot ";
 217                         } else {
 218                                 altroot = "";
 219                         }

 221                         if (fcn == AD_FASTREBOOT) {
 222                                 char *newarg, *head;
```



```
 223                                 char bargs_scratch[BOOTARGS_MAX];

 225                                 bzero(bargs_scratch, BOOTARGS_MAX);

 227                                 bcopy(bargs, bargs_scratch, strlen(bargs));
 228                                 head = bargs_scratch;
 229                                 newarg = strtok(bargs_scratch, " ");

 231                                 if (newarg == NULL || newarg[0] == '-')
 232                                         break;

 234                                 /* First argument is rootdir */
 235                                 if (strncmp(&newarg[strlen(newarg)-4],
 236                                     "unix", 4) != 0) {
 237                                         newarg = strtok(NULL, " ");
 238                                         off = newarg - head;
 239                                 }

 241                                 /*
 242                                  * If we are using alternate root via
 243                                  * mountpoint or a different BE, don't
 244                                  * bother to update the temp menu entry.
 245                                  */
 246                                 if (off > 0)
 247                                         break;
 248                         }

 250                         /* are we rebooting to a GRUB menu entry? */
 251                         if (isdigit(bargs[0])) {
 252                                 int entry = strtol(bargs, NULL, 10);
 253                                 (void) snprintf(cmdbuf, sizeof (cmdbuf),
 254                                     "/sbin/grubadm %s --set-default %d",
 254                                     "/sbin/bootadm set-menu %sdefault=%d",
 255                                     altroot, entry);
 256                         } else {
 257                                 (void) snprintf(cmdbuf, sizeof (cmdbuf),
 258                                     "/sbin/grubadm --new --default %s"
 259                                     "--set-opts %s%s%s", altroot, quote,
 258                                     "/sbin/bootadm -m update_temp %s"
 259                                     "-o %s%s%s", altroot, quote,
 260                                     &bargs[off], quote);
 261                         }
 262                         (void) system(cmdbuf);
 263                 }
 264                 check_archive_update();
 265         }

 267         return (__uadmin(cmd, fcn, mdep));
 268 }
______unchanged_portion_omitted_
```

```
**********************************************************
    7186 Fri Jul 27 03:18:39 2012
new/usr/src/lib/libc/i386/sys/uadmin.c
uadmin
**********************************************************
______unchanged_portion_omitted_

 164 static char quote[] = "\'";

 166 int
 167 uadmin(int cmd, int fcn, uintptr_t mdep)
 168 {
 169         extern int __uadmin(int cmd, int fcn, uintptr_t mdep);
 170         char *bargs, cmdbuf[256];
 171         struct stat sbuf;
 172         char *altroot;

 174         bargs = (char *)mdep;

 176         if (geteuid() == 0 && getzoneid() == GLOBAL_ZONEID &&
 177             (cmd == A_SHUTDOWN || cmd == A_REBOOT)) {
 178                 int off = 0;

 180                 switch (fcn) {
 181                 case AD_IBOOT:
 182                 case AD_SBOOT:
 183                 case AD_SIBOOT:
 184                         /*
 185                          * These functions fabricate appropriate bootargs.
 186                          * If bootargs are passed in, map these functions
 187                          * to AD_BOOT.
 188                          */
 189                         if (bargs == 0) {
 190                                 switch (fcn) {
 191                                 case AD_IBOOT:
 192                                         bargs = "-a";
 193                                         break;
 194                                 case AD_SBOOT:
 195                                         bargs = "-s";
 196                                         break;
 197                                 case AD_SIBOOT:
 198                                         bargs = "-sa";
 199                                         break;
 200                                 }
 201                         }
 202                         /*FALLTHROUGH*/
 203                 case AD_BOOT:
 204                 case AD_FASTREBOOT:
 205                         if (bargs == 0)
 206                                 break;  /* no args */
 207                         if (legal_arg(bargs) < 0)
 208                                 break;  /* bad args */

 210                         /* avoid cancellation in system() */
 211                         (void) pthread_setcancelstate(PTHREAD_CANCEL_DISABLE,
 212                             NULL);

 214                         /* check for /stubboot */
 215                         if (stat("/stubboot/boot/grub/menu.lst", &sbuf) == 0) {
 216                                 altroot = "-R /stubboot ";
 217                         } else {
 218                                 altroot = "";
 219                         }

 221                         if (fcn == AD_FASTREBOOT) {
 222                                 char *newarg, *head;
```



```
 223                                 char bargs_scratch[BOOTARGS_MAX];

 225                                 bzero(bargs_scratch, BOOTARGS_MAX);

 227                                 bcopy(bargs, bargs_scratch, strlen(bargs));
 228                                 head = bargs_scratch;
 229                                 newarg = strtok(bargs_scratch, " ");

 231                                 if (newarg == NULL || newarg[0] == '-')
 232                                         break;

 234                                 /* First argument is rootdir */
 235                                 if (strncmp(&newarg[strlen(newarg)-4],
 236                                     "unix", 4) != 0) {
 237                                         newarg = strtok(NULL, " ");
 238                                         off = newarg - head;
 239                                 }

 241                                 /*
 242                                  * If we are using alternate root via
 243                                  * mountpoint or a different BE, don't
 244                                  * bother to update the temp menu entry.
 245                                  */
 246                                 if (off > 0)
 247                                         break;
 248                         }

 250                         /* are we rebooting to a GRUB menu entry? */
 251                         if (isdigit(bargs[0])) {
 252                                 int entry = strtol(bargs, NULL, 10);
 253                                 (void) snprintf(cmdbuf, sizeof (cmdbuf),
 254                                     "/sbin/grubadm %s --set-default %d",
 254                                     "/sbin/bootadm set-menu %sdefault=%d",
 255                                     altroot, entry);
 256                         } else {
 257                                 (void) snprintf(cmdbuf, sizeof (cmdbuf),
 258                                     "/sbin/grubadm --new --default %s"
 259                                     "--set-opts %s%s%s", altroot, quote,
 258                                     "/sbin/bootadm -m update_temp %s"
 259                                     "-o %s%s%s", altroot, quote,
 260                                     &bargs[off], quote);
 261                         }
 262                         (void) system(cmdbuf);
 263                 }
 264                 check_archive_update();
 265         }
 266         return (__uadmin(cmd, fcn, mdep));
 267 }
______unchanged_portion_omitted_
```