

```

*****
52404 Mon May 5 01:02:55 2014
new/usr/src/cmd/hal/hald/solaris/devinfo_storage.c
4846 HAL partition names don't match real partition names
*****
1 /*****
2 *
3 * devinfo_storage.c : storage devices
4 *
5 * Copyright (c) 2006, 2010, Oracle and/or its affiliates. All rights reserved.
6 * Copyright 2013 Garrett D'Amore <garrett@damore.org>
7 * Copyright 2014 Andrew Stormont.
8 #endif /* ! codereview */
9 *
10 * Licensed under the Academic Free License version 2.1
11 *
12 *****/

14 #ifdef HAVE_CONFIG_H
15 # include <config.h>
16 #endif

18 #include <stdio.h>
19 #include <string.h>
20 #include <strings.h>
21 #include <ctype.h>
22 #include <libdevinfo.h>
23 #include <sys/types.h>
24 #include <sys/mkdev.h>
25 #include <sys/stat.h>
26 #include <sys/mntent.h>
27 #include <sys/mnttab.h>

29 #include "../osspec.h"
30 #include "../logger.h"
31 #include "../hald.h"
32 #include "../hald_dbus.h"
33 #include "../device_info.h"
34 #include "../util.h"
35 #include "../hald_runner.h"
36 #include "hotplug.h"
37 #include "devinfo.h"
38 #include "devinfo_misc.h"
39 #include "devinfo_storage.h"
40 #include "osspec_solaris.h"

42 #ifdef sparc
43 #define WHOLE_DISK "s2"
44 #define DOS_SEPERATOR ":"
45 #define DOS_FORMAT "s2:%d"
46 #define DOS_TEMPLATE ":NN"
47 #endif /* ! codereview */
48 #else
49 #define WHOLE_DISK "p0"
50 #define DOS_SEPERATOR "p"
51 #define DOS_FORMAT "p%d"
52 #define DOS_TEMPLATE "pNN"
53 #endif /* ! codereview */
54 #endif

56 /* some devices, especially CDROMs, may take a while to be probed (values in ms)
57 #define DEVINFO_PROBE_STORAGE_TIMEOUT 60000
58 #define DEVINFO_PROBE_VOLUME_TIMEOUT 60000

60 typedef struct devinfo_storage_minor {
61     char *devpath;

```

```

62     char *devlink;
63     char *slice;
64     dev_t dev;
65     int dosnum; /* dos disk number or -1 */
66 } devinfo_storage_minor_t;

68 HalDevice *devinfo_ide_add(HalDevice *parent, di_node_t node, char *devfs_path,
69 static HalDevice *devinfo_ide_host_add(HalDevice *parent, di_node_t node, char *
70 static HalDevice *devinfo_ide_device_add(HalDevice *parent, di_node_t node, char
71 static HalDevice *devinfo_ide_storage_add(HalDevice *parent, di_node_t node, cha
72 HalDevice *devinfo_scsi_add(HalDevice *parent, di_node_t node, char *devfs_path,
73 static HalDevice *devinfo_scsi_storage_add(HalDevice *parent, di_node_t node, ch
74 HalDevice *devinfo_blkdev_add(HalDevice *parent, di_node_t node, char *devfs_pat
75 static HalDevice *devinfo_blkdev_storage_add(HalDevice *parent, di_node_t node,
76 HalDevice *devinfo_floppy_add(HalDevice *parent, di_node_t node, char *devfs_pat
77 static void devinfo_floppy_add_volume(HalDevice *parent, di_node_t node);
78 static HalDevice *devinfo_lofi_add(HalDevice *parent, di_node_t node, char *devf
79 static void devinfo_lofi_add_minor(HalDevice *parent, di_node_t node, char *mino
80 static void devinfo_storage_minors(HalDevice *parent, di_node_t node, gchar *dev
81 static struct devinfo_storage_minor *devinfo_storage_new_minor(char *maindev_pat
82     char *devlink, dev_t dev, int dosnum);
83 static void devinfo_storage_free_minor(struct devinfo_storage_minor *m);
84 HalDevice *devinfo_volume_add(HalDevice *parent, di_node_t node, devinfo_storage
85 static void devinfo_volume_preprobing_done(HalDevice *d, gpointer userdata, gpo
86 static void devinfo_volume_hotplug_begin_add (HalDevice *d, HalDevice *parent, D
87 static void devinfo_storage_hotplug_begin_add (HalDevice *d, HalDevice *parent,
88 static void devinfo_storage_probing_done (HalDevice *d, guint32 exit_type, gint
89 const gchar *devinfo_volume_get_prober (HalDevice *d, int *timeout);
90 const gchar *devinfo_storage_get_prober (HalDevice *d, int *timeout);

92 static char *devinfo_scsi_dtype2str(int dtype);
93 static char *devinfo_volume_get_slice_name (char *devlink);
94 static gboolean dos_to_dev(char *path, char **devpath, int *partnum);
95 static gboolean is_dos_path(char *path, int *partnum);

97 static void devinfo_storage_set_nicknames (HalDevice *d);

99 DevinfoDevHandler devinfo_ide_handler = {
100     devinfo_ide_add,
101     NULL,
102     NULL,
103     NULL,
104     NULL,
105     NULL
106 };
107 DevinfoDevHandler devinfo_scsi_handler = {
108     devinfo_scsi_add,
109     NULL,
110     NULL,
111     NULL,
112     NULL,
113     NULL
114 };
115 DevinfoDevHandler devinfo_blkdev_handler = {
116     devinfo_blkdev_add,
117     NULL,
118     NULL,
119     NULL,
120     NULL,
121     NULL
122 };
123 DevinfoDevHandler devinfo_floppy_handler = {
124     devinfo_floppy_add,
125     NULL,
126     NULL,
127     NULL,

```

```

128     NULL,
129     NULL
130 };
131 DevinfoDevHandler devinfo_lofi_handler = {
132     devinfo_lofi_add,
133     NULL,
134     NULL,
135     NULL,
136     NULL,
137     NULL
138 };
139 DevinfoDevHandler devinfo_storage_handler = {
140     NULL,
141     NULL,
142     devinfo_storage_hotplug_begin_add,
143     NULL,
144     devinfo_storage_probing_done,
145     devinfo_storage_get_prober
146 };
147 DevinfoDevHandler devinfo_volume_handler = {
148     NULL,
149     NULL,
150     devinfo_volume_hotplug_begin_add,
151     NULL,
152     NULL,
153     devinfo_volume_get_prober
154 };

156 /* IDE */

158 HalDevice *
159 devinfo_ide_add(HalDevice *parent, di_node_t node, char *devfs_path, char *device)
160 {
161     char *s;

163     if ((device_type != NULL) && (strcmp(device_type, "ide") == 0)) {
164         return (devinfo_ide_host_add(parent, node, devfs_path));
165     }

167     if ((di_prop_lookup_strings (DDI_DEV_T_ANY, node, "class", &s) > 0) &&
168         (strcmp (s, "dada") == 0)) {
169         return (devinfo_ide_device_add(parent, node, devfs_path));
170     }

172     return (NULL);
173 }

175 static HalDevice *
176 devinfo_ide_host_add(HalDevice *parent, di_node_t node, char *devfs_path)
177 {
178     HalDevice *d;

180     d = hal_device_new ();

182     devinfo_set_default_properties (d, parent, node, devfs_path);
183     hal_device_property_set_string (d, "info.product", "IDE host controller");
184     hal_device_property_set_string (d, "info.subsystem", "ide_host");
185     hal_device_property_set_int (d, "ide_host.number", 0); /* XXX */

187     devinfo_add_enqueue (d, devfs_path, &devinfo_ide_handler);

189     return (d);
190 }

192 static HalDevice *
193 devinfo_ide_device_add(HalDevice *parent, di_node_t node, char *devfs_path)

```

```

194 {
195     HalDevice *d;

197     d = hal_device_new();

199     devinfo_set_default_properties (d, parent, node, devfs_path);
200     hal_device_property_set_string (parent, "info.product", "IDE device");
201     hal_device_property_set_string (parent, "info.subsystem", "ide");
202     hal_device_property_set_int (parent, "ide.host", 0); /* XXX */
203     hal_device_property_set_int (parent, "ide.channel", 0);

205     devinfo_add_enqueue (d, devfs_path, &devinfo_ide_handler);

207     return (devinfo_ide_storage_add (d, node, devfs_path));
208 }

210 static HalDevice *
211 devinfo_ide_storage_add(HalDevice *parent, di_node_t node, char *devfs_path)
212 {
213     HalDevice *d;
214     char *s;
215     int *i;
216     char *driver_name;
217     char udi[HAL_PATH_MAX];

219     if ((driver_name = di_driver_name (node)) == NULL) {
220         return (NULL);
221     }

223     d = hal_device_new ();

225     devinfo_set_default_properties (d, parent, node, devfs_path);
226     hal_device_property_set_string (d, "info.category", "storage");

228     hal_util_compute_udi (hald_get_gdl (), udi, sizeof (udi),
229         "%s/%s%d", hal_device_get_udi (parent), driver_name, di_instance);
230     hal_device_set_udi (d, udi);
231     hal_device_property_set_string (d, "info.udi", udi);
232     PROP_STR(d, node, s, "devid", "info.product");

234     hal_device_add_capability (d, "storage");
235     hal_device_property_set_string (d, "storage.bus", "ide");
236     hal_device_property_set_int (d, "storage.lun", 0);
237     hal_device_property_set_string (d, "storage.drive_type", "disk");

239     PROP_BOOL(d, node, i, "hotpluggable", "storage.hotpluggable");
240     PROP_BOOL(d, node, i, "removable-media", "storage.removable");

242     hal_device_property_set_bool (d, "storage.media_check_enabled", FALSE);

244     /* XXX */
245     hal_device_property_set_bool (d, "storage.requires_eject", FALSE);

247     hal_device_add_capability (d, "block");

249     devinfo_storage_minors (d, node, (char *)devfs_path, FALSE);

251     return (d);
252 }

254 /* SCSI */

256 HalDevice *
257 devinfo_scsi_add(HalDevice *parent, di_node_t node, char *devfs_path, char *device)
258 {
259     int *i;

```

```

260     char    *driver_name;
261     HalDevice *d;
262     char    udi[HAL_PATH_MAX];

264     driver_name = di_driver_name (node);
265     if ((driver_name == NULL) || (strcmp (driver_name, "sd") != 0)) {
266         return (NULL);
267     }

269     d = hal_device_new ();

271     devinfo_set_default_properties (d, parent, node, devfs_path);
272     hal_device_property_set_string (d, "info.subsystem", "scsi");

274     hal_util_compute_udi (hald_get_gdl (), udi, sizeof (udi),
275                          "%s/%s%d", hal_device_get_udi (parent), di_node_name(node), di_i
276     hal_device_set_udi (d, udi);
277     hal_device_property_set_string (d, "info.udi", udi);

279     hal_device_property_set_int (d, "scsi.host",
280                                hal_device_property_get_int (parent, "scsi_host.host"));
281     hal_device_property_set_int (d, "scsi.bus", 0);
282     PROP_INT(d, node, i, "target", "scsi.target");
283     PROP_INT(d, node, i, "lun", "scsi.lun");
284     hal_device_property_set_string (d, "info.product", "SCSI Device");

286     devinfo_add_enqueue (d, devfs_path, &devinfo_scsi_handler);

288     return (devinfo_scsi_storage_add (d, node, devfs_path));
289 }

291 static HalDevice *
292 devinfo_scsi_storage_add(HalDevice *parent, di_node_t node, char *devfs_path)
293 {
294     HalDevice *d;
295     int *i;
296     char *s;
297     char    udi[HAL_PATH_MAX];

299     d = hal_device_new ();

301     devinfo_set_default_properties (d, parent, node, devfs_path);
302     hal_device_property_set_string (d, "info.category", "storage");

304     hal_util_compute_udi (hald_get_gdl (), udi, sizeof (udi),
305                          "%s/sd%d", hal_device_get_udi (parent), di_instance (node));
306     hal_device_set_udi (d, udi);
307     hal_device_property_set_string (d, "info.udi", udi);
308     PROP_STR(d, node, s, "inquiry-product-id", "info.product");

310     hal_device_add_capability (d, "storage");

312     hal_device_property_set_int (d, "storage.lun",
313                                 hal_device_property_get_int (parent, "scsi.lun"));
314     PROP_BOOL(d, node, i, "hotpluggable", "storage.hotpluggable");
315     PROP_BOOL(d, node, i, "removable-media", "storage.removable");
316     hal_device_property_set_bool (d, "storage.requires_eject", FALSE);

318     /*
319     * We have to enable polling not only for drives with removable media,
320     * but also for hotpluggable devices, because when a disk is
321     * unplugged while busy/mounted, there is not sysevent generated.
322     * Instead, the HBA driver (scsa2usb, scsa1394) will notify sd driver
323     * and the latter will report DKIO_DEV_GONE via DKIOCSTATE ioctl.
324     * So we have to enable media check so that hald-addon-storage notices
325     * the "device gone" condition and unmounts all associated volumes.

```

```

326     /*
327     hal_device_property_set_bool (d, "storage.media_check_enabled",
328                                  ((di_prop_lookup_ints(DDI_DEV_T_ANY, node, "removable-media", &i) >=
329                                   (di_prop_lookup_ints(DDI_DEV_T_ANY, node, "hotpluggable", &i) >= 0))

331     if (di_prop_lookup_ints(DDI_DEV_T_ANY, node, "inquiry-device-type",
332                             &i) > 0) {
333         s = devinfo_scsi_dtype2str (*i);
334         hal_device_property_set_string (d, "storage.drive_type", s);

336         if (strcmp (s, "cdrom") == 0) {
337             hal_device_add_capability (d, "storage.cdrom");
338             hal_device_property_set_bool (d, "storage.no_partitions_
339             hal_device_property_set_bool (d, "storage.requires_eject
340         }
341     }

343     hal_device_add_capability (d, "block");

345     devinfo_storage_minors (d, node, devfs_path, FALSE);

347     return (d);
348 }

350 static char *
351 devinfo_scsi_dtype2str(int dtype)
352 {
353     char *dtype2str[] = {
354         "disk" ,           /* DTYPE_DIRECT           0x00 */
355         "tape" ,          /* DTYPE_SEQUENTIAL      0x01 */
356         "printer" ,       /* DTYPE_PRINTER         0x02 */
357         "processor" ,     /* DTYPE_PROCESSOR       0x03 */
358         "worm" ,          /* DTYPE_WORM            0x04 */
359         "cdrom" ,         /* DTYPE_RODIRECT        0x05 */
360         "scanner" ,      /* DTYPE_SCANNER         0x06 */
361         "cdrom" ,         /* DTYPE_OPTICAL         0x07 */
362         "changer" ,      /* DTYPE_CHANGER         0x08 */
363         "comm" ,          /* DTYPE_COMM            0x09 */
364         "scsi" ,          /* DTYPE_???             0x0A */
365         "scsi" ,          /* DTYPE_???             0x0B */
366         "array_ctrl" ,   /* DTYPE_ARRAY_CTRL     0x0C */
367         "esi" ,           /* DTYPE_ESI             0x0D */
368         "disk" ,         /* DTYPE_RBC             0x0E */
369     };

371     if (dtype < NELEM(dtype2str)) {
372         return (dtype2str[dtype]);
373     } else {
374         return ("scsi");
375     }

377 }

379 /* blkdev */

381 HalDevice *
382 devinfo_blkdev_add(HalDevice *parent, di_node_t node, char *devfs_path, char *de
383 {
384     int *i;
385     char *driver_name;
386     HalDevice *d;
387     char    udi[HAL_PATH_MAX];

389     driver_name = di_driver_name (node);
390     if ((driver_name == NULL) || (strcmp (driver_name, "blkdev") != 0)) {
391         return (NULL);

```

```

392     }
394     d = hal_device_new ();

396     devinfo_set_default_properties (d, parent, node, devfs_path);
397     hal_device_property_set_string (d, "info.subsystem", "pseudo");

399     hal_util_compute_udi (hald_get_gdl (), udi, sizeof (udi),
400                          "%s/%s%d", hal_device_get_udi (parent), di_node_name(node), di_i
401                          hal_device_set_udi (d, udi);
402     hal_device_property_set_string (d, "info.udi", udi);
403     hal_device_property_set_string (d, "info.product", "Block Device");

405     devinfo_add_enqueue (d, devfs_path, &devinfo_blkdev_handler);

407     return (devinfo_blkdev_storage_add (d, node, devfs_path));
408 }

410 static HalDevice *
411 devinfo_blkdev_storage_add(HalDevice *parent, di_node_t node, char *devfs_path)
412 {
413     HalDevice *d;
414     char *driver_name;
415     int *i;
416     char *s;
417     char udi[HAL_PATH_MAX];

419     d = hal_device_new ();

421     devinfo_set_default_properties (d, parent, node, devfs_path);
422     hal_device_property_set_string (d, "info.category", "storage");

424     hal_util_compute_udi (hald_get_gdl (), udi, sizeof (udi),
425                          "%s/blkdev%d", hal_device_get_udi (parent), di_instance (node));
426     hal_device_set_udi (d, udi);
427     hal_device_property_set_string (d, "info.udi", udi);

429     hal_device_add_capability (d, "storage");

431     hal_device_property_set_int (d, "storage.lun", 0);

433     PROP_BOOL(d, node, i, "hotpluggable", "storage.hotpluggable");
434     PROP_BOOL(d, node, i, "removable-media", "storage.removable");

436     hal_device_property_set_bool (d, "storage.requires_eject", FALSE);
437     hal_device_property_set_bool (d, "storage.media_check_enabled", TRUE);
438     hal_device_property_set_string (d, "storage.drive_type", "disk");

440     hal_device_add_capability (d, "block");

442     devinfo_storage_minors (d, node, devfs_path, FALSE);

444     return (d);
445 }

447 /* floppy */

449 HalDevice *
450 devinfo_floppy_add(HalDevice *parent, di_node_t node, char *devfs_path, char *de
451 {
452     char *driver_name;
453     char *raw;
454     char udi[HAL_PATH_MAX];
455     di_devlink_handle_t devlink_hdl;
456     int major;
457     di_minor_t minor;

```

```

458     dev_t dev;
459     HalDevice *d = NULL;
460     char *minor_path = NULL;
461     char *devlink = NULL;

463     driver_name = di_driver_name (node);
464     if ((driver_name == NULL) || (strcmp (driver_name, "fd") != 0)) {
465         return (NULL);
466     }

468     /*
469     * The only minor node we're interested in is /dev/diskette*
470     */
471     major = di_driver_major(node);
472     if ((devlink_hdl = di_devlink_init(NULL, 0)) == NULL) {
473         return (NULL);
474     }
475     minor = DI_MINOR_NIL;
476     while ((minor = di_minor_next(node, minor)) != DI_MINOR_NIL) {
477         dev = di_minor_devt(minor);
478         if ((major != major(dev)) ||
479             (di_minor_type(minor) != DDM_MINOR) ||
480             (di_minor_spectype(minor) != S_IFBLK) ||
481             ((minor_path = di_devfs_minor_path(minor)) == NULL)) {
482             continue;
483         }
484         if ((devlink = get_devlink(devlink_hdl, "diskette.+ ", minor_pat
485             break;
486     }
487     di_devfs_path_free (minor_path);
488     minor_path = NULL;
489     free(devlink);
490     devlink = NULL;
491 }
492 di_devlink_fini (&devlink_hdl);

494     if ((devlink == NULL) || (minor_path == NULL)) {
495         HAL_INFO (("floppy devlink not found %s", devfs_path));
496         goto out;
497     }

499     d = hal_device_new ();

501     devinfo_set_default_properties (d, parent, node, devfs_path);
502     hal_device_property_set_string (d, "info.category", "storage");
503     hal_device_add_capability (d, "storage");
504     hal_device_property_set_string (d, "storage.bus", "platform");
505     hal_device_property_set_bool (d, "storage.hotpluggable", FALSE);
506     hal_device_property_set_bool (d, "storage.removable", TRUE);
507     hal_device_property_set_bool (d, "storage.requires_eject", TRUE);
508     hal_device_property_set_bool (d, "storage.media_check_enabled", FALSE);
509     hal_device_property_set_string (d, "storage.drive_type", "floppy");

511     hal_device_add_capability (d, "block");
512     hal_device_property_set_bool (d, "block.is_volume", FALSE);
513     hal_device_property_set_int (d, "block.major", major(dev));
514     hal_device_property_set_int (d, "block.minor", minor(dev));
515     hal_device_property_set_string (d, "block.device", devlink);
516     raw = dsk_to_rdisk (devlink);
517     hal_device_property_set_string (d, "block.solaris.raw_device", raw);
518     free (raw);

520     devinfo_add_enqueue (d, devfs_path, &devinfo_storage_handler);

522     /* trigger initial probe-volume */
523     devinfo_floppy_add_volume(d, node);

```

```

525 out:
526     di_devfs_path_free (minor_path);
527     free(devlink);
529     return (d);
530 }

532 static void
533 devinfo_floppy_add_volume(HalDevice *parent, di_node_t node)
534 {
535     char    *devlink;
536     char    *devfs_path;
537     int     minor, major;
538     dev_t   dev;
539     struct devinfo_storage_minor *m;

541     devfs_path = (char *)hal_device_property_get_string (parent, "solaris.de
542     devlink = (char *)hal_device_property_get_string (parent, "block.device"
543     major = hal_device_property_get_int (parent, "block.major");
544     minor = hal_device_property_get_int (parent, "block.minor");
545     dev = makedev (major, minor);

547     m = devinfo_storage_new_minor (devfs_path, WHOLE_DISK, devlink, dev, -1)
548     devinfo_volume_add (parent, node, m);
549     devinfo_storage_free_minor (m);
550 }

552 /*
553  * After reprobng storage, reprobe its volumes.
554  */
555 static void
556 devinfo_floppy_rescan_probing_done (HalDevice *d, quint32 exit_type, gint return
557     char **error, gpointer userdata1, gpointer userdata2)
558 {
559     void *end_token = (void *) userdata1;
560     const char *devfs_path;
561     di_node_t node;
562     HalDevice *v;

564     if (!hal_device_property_get_bool (d, "storage.removable.media_available
565         HAL_INFO (("no floppy media", hal_device_get_udi (d)));

567         /* remove child (can only be single volume) */
568         if (((v = hal_device_store_match_key_value_string (hald_get_gdl(
569             "info.parent", hal_device_get_udi (d))) != NULL) &&
570             ((devfs_path = hal_device_property_get_string (v,
571                 "solaris.devfs_path") != NULL)) {
572             devinfo_remove_enqueue ((char *)devfs_path, NULL);
573         }
574     } else {
575         HAL_INFO (("floppy media found", hal_device_get_udi (d));

577         if ((devfs_path = hal_device_property_get_string(d, "solaris.dev
578             HAL_INFO (("no devfs_path", hal_device_get_udi (d)));
579             hotplug_event_process_queue ();
580         }
581     }
582     if ((node = di_init (devfs_path, DINFOCPYALL)) == DI_NODE_NIL) {
583         HAL_INFO (("di_init %s failed %d", devfs_path, errno));
584         hotplug_event_process_queue ();
585     }
586 }

588     devinfo_floppy_add_volume (d, node);

```

```

590         di_fini (node);
591     }

593     hotplug_event_process_queue ();
594 }
595
596 /* lofi */

598 HalDevice *
599 devinfo_lofi_add(HalDevice *parent, di_node_t node, char *devfs_path, char *devi
600 {
601     return (devinfo_lofi_add_major(parent,node, devfs_path, device_type, FAL
602 }

604 HalDevice *
605 devinfo_lofi_add_major(HalDevice *parent, di_node_t node, char *devfs_path, char
606     gboolean rescan, HalDevice *lofi_d)
607 {
608     char    *driver_name;
609     HalDevice *d = NULL;
610     char    udi[HAL_PATH_MAX];
611     di_devlink_handle_t devlink_hdl;
612     int     major;
613     di_minor_t minor;
614     dev_t   dev;
615     char    *minor_path = NULL;
616     char    *devlink = NULL;

618     driver_name = di_driver_name (node);
619     if ((driver_name == NULL) || (strcmp (driver_name, "lofi") != 0)) {
620         return (NULL);
621     }

623     if (!rescan) {
624         d = hal_device_new ();

626         devinfo_set_default_properties (d, parent, node, devfs_path);
627         hal_device_property_set_string (d, "info.subsystem", "pseudo");

629         hal_util_compute_udi (hald_get_gdl (), udi, sizeof (udi),
630             "%s/%s%d", hal_device_get_udi (parent), di_node_name(nod
631         hal_device_set_udi (d, udi);
632         hal_device_property_set_string (d, "info.udi", udi);

634         devinfo_add_enqueue (d, devfs_path, &devinfo_lofi_handler);
635     } else {
636         d = lofi_d;
637     }

639     /*
640     * Unlike normal storage, as in devinfo_storage_minors(), where
641     * sd instance -> HAL storage, sd minor node -> HAL volume,
642     * lofi always has one instance, lofi minor -> HAL storage.
643     * lofi storage never has slices, but it can have
644     * embedded pcfs partitions that fstyp would recognize
645     */
646     major = di_driver_major(node);
647     if ((devlink_hdl = di_devlink_init(NULL, 0)) == NULL) {
648         return (d);
649     }
650     minor = DI_MINOR_NIL;
651     while ((minor = di_minor_next(node, minor)) != DI_MINOR_NIL) {
652         dev = di_minor_devt(minor);
653         if ((major != major(dev)) ||
654             (di_minor_type(minor) != DDM_MINOR) ||
655             (di_minor_spectype(minor) != S_IFBLK) ||

```

```

656         ((minor_path = di_devfs_minor_path(minor)) == NULL) {
657             continue;
658         }
659         if ((devlink = get_devlink(devlink_hdl, NULL, minor_path)) == NU
660             di_devfs_path_free (minor_path);
661             continue;
662         }
663     }
664     if (!rescan ||
665         (hal_device_store_match_key_value_string (hald_get_gdl (),
666           "solaris.devfs_path", minor_path) == NULL)) {
667         devinfo_lofi_add_minor(d, node, minor_path, devlink, dev
668     }
669     di_devfs_path_free (minor_path);
670     free(devlink);
671 }
672 di_devlink_fini (&devlink_hdl);
673
674 return (d);
675 }
676
677 static void
678 devinfo_lofi_add_minor(HalDevice *parent, di_node_t node, char *minor_path, char
679 {
680     HalDevice *d;
681     char *raw;
682     char *doslink;
683     char *dospath[64];
684     struct devinfo_storage_minor *m;
685     int i;
686
687     /* add storage */
688     d = hal_device_new ();
689
690     devinfo_set_default_properties (d, parent, node, minor_path);
691     hal_device_property_set_string (d, "info.category", "storage");
692     hal_device_add_capability (d, "storage");
693     hal_device_property_set_string (d, "storage.bus", "lofi");
694     hal_device_property_set_bool (d, "storage.hotpluggable", TRUE);
695     hal_device_property_set_bool (d, "storage.removable", FALSE);
696     hal_device_property_set_bool (d, "storage.requires_eject", FALSE);
697     hal_device_property_set_string (d, "storage.drive_type", "disk");
698     hal_device_add_capability (d, "block");
699     hal_device_property_set_int (d, "block.major", major(dev));
700     hal_device_property_set_int (d, "block.minor", minor(dev));
701     hal_device_property_set_string (d, "block.device", devlink);
702     raw = dsk_to_rdisk (devlink);
703     hal_device_property_set_string (d, "block.solaris.raw_device", raw);
704     free (raw);
705     hal_device_property_set_bool (d, "block.is_volume", FALSE);
706
707     devinfo_add_enqueue (d, minor_path, &devinfo_storage_handler);
708
709     /* add volumes: one on main device and a few pcfs candidates */
710     m = devinfo_storage_new_minor(minor_path, WHOLE_DISK, devlink, dev, -1);
711     devinfo_volume_add (d, node, m);
712     devinfo_storage_free_minor (m);
713
714     doslink = (char *)calloc (1, strlen (devlink) + sizeof (DOS_TEMPLATE) +
715     doslink = (char *)calloc (1, strlen (devlink) + sizeof ("::NNN") + 1);
716     if (doslink != NULL) {
717         for (i = 1; i < 16; i++) {
718             snprintf(dospath, sizeof (dospath), DOS_FORMAT, i);
719             sprintf(doslink, "%sDOS_SEPERATOR%d", devlink, i);
720             snprintf(dospath, sizeof (dospath), WHOLE_DISK"%d", i);

```

```

11         sprintf(doslink, "%s:%d", devlink, i);
120         m = devinfo_storage_new_minor(minor_path, dospath, dosli
121         devinfo_volume_add (d, node, m);
122         devinfo_storage_free_minor (m);
123     }
124     free (doslink);
125 }
126 }
    unchanged_portion_omitted
805 /*
806  * Storage minor nodes are potential "volume" objects.
807  * This function also completes building the parent object (main storage device)
808  */
809 static void
810 devinfo_storage_minors(HalDevice *parent, di_node_t node, gchar *devfs_path, gbo
811 {
812     di_devlink_handle_t devlink_hdl;
813     gboolean is_cdrom;
814     const char *whole_disk;
815     int major;
816     di_minor_t minor;
817     dev_t dev;
818     char *minor_path = NULL;
819     char *maindev_path = NULL;
820     char *devpath, *devlink;
821     int doslink_len;
822     char *doslink;
823     char dospath[64];
824     char *slice;
825     int pathlen;
826     int i;
827     char *raw;
828     boolean_t maindev_is_d0;
829     GQueue *mq;
830     HalDevice *volume;
831     struct devinfo_storage_minor *m;
832     struct devinfo_storage_minor *maindev = NULL;
833
834     /* for cdroms whole disk is always s2 */
835     is_cdrom = hal_device_has_capability (parent, "storage.cdrom");
836     whole_disk = is_cdrom ? "s2" : WHOLE_DISK;
837
838     major = di_driver_major(node);
839
840     /* the "whole disk" p0/s2/d0 node must come first in the hotplug queue
841     * so we put other minor nodes on the local queue and move to the
842     * hotplug queue up in the end
843     */
844     if ((mq = g_queue_new()) == NULL) {
845         goto err;
846     }
847     if ((devlink_hdl = di_devlink_init(NULL, 0)) == NULL) {
848         g_queue_free (mq);
849         goto err;
850     }
851     minor = DI_MINOR_NIL;
852     while ((minor = di_minor_next(node, minor)) != DI_MINOR_NIL) {
853         dev = di_minor_devt(minor);
854         if ((major != major(dev)) ||
855             (di_minor_type(minor) != DDM_MINOR) ||
856             (di_minor_spectype(minor) != S_IFBLK) ||
857             ((minor_path = di_devfs_minor_path(minor)) == NULL)) {
858             continue;
859         }
860         if ((devlink = get_devlink(devlink_hdl, NULL, minor_path)) == NU

```

```

861         di_devfs_path_free (minor_path);
862         continue;
863     }

865     slice = devinfo_volume_get_slice_name (devlink);
866     if (strlen (slice) < 2) {
867         free (devlink);
868         di_devfs_path_free (minor_path);
869         continue;
870     }

164     /* ignore p1..N - we'll use p0:N instead */
165     if ((strlen (slice) > 1) && (slice[0] == 'p') && isdigit(slice[1]
166         ((atol(&slice[1])) > 0)) {
167         free (devlink);
168         di_devfs_path_free (minor_path);
169         continue;
170     }

872     m = devinfo_storage_new_minor(minor_path, slice, devlink, dev, -
873     if (m == NULL) {
874         free (devlink);
875         di_devfs_path_free (minor_path);
876         continue;
877     }

879     /* main device is either s2/p0 or d0, the latter taking preceden
880     if ((strcmp (slice, "d0") == 0) ||
881         ((strcmp (slice, whole_disk) == 0) && (maindev == NULL))))
882         if (maindev_path != NULL) {
883             di_devfs_path_free (maindev_path);
884         }
885         maindev_path = minor_path;
886         maindev = m;
887         g_queue_push_head (mq, maindev);
888     } else {
889         di_devfs_path_free (minor_path);
890         g_queue_push_tail (mq, m);
891     }

893     free (devlink);
894 }
895 di_devlink_fini (&devlink_hdl);

897 if (maindev == NULL) {
898     /* shouldn't typically happen */
899     while (!g_queue_is_empty (mq)) {
900         devinfo_storage_free_minor (g_queue_pop_head (mq));
901     }
902     goto err;
903 }

905 /* first enqueue main storage device */
906 if (!rescan) {
907     hal_device_property_set_int (parent, "block.major", major);
908     hal_device_property_set_int (parent, "block.minor", minor(mainde
909     hal_device_property_set_string (parent, "block.device", maindev-
910     raw = dsk_to_rdisk (maindev->devlink);
911     hal_device_property_set_string (parent, "block.solaris.raw_devic
912     free (raw);
913     hal_device_property_set_bool (parent, "block.is_volume", FALSE);
914     hal_device_property_set_string (parent, "solaris.devfs_path", ma
915     devinfo_add_enqueue (parent, maindev_path, &devinfo_storage_hand
916 }

918 /* add virtual dos volumes to enable pcfs probing */

```

```

919     if (!is_cdrom) {
920         doslink_len = strlen (maindev->devlink) + sizeof (DOS_TEMPLATE)
921         doslink_len = strlen (maindev->devlink) + sizeof (":NNN") + 1;
922         if ((doslink = (char *)calloc (1, doslink_len)) != NULL) {
923             for (i = 1; i < 16; i++) {
924                 snprintf(dospath, sizeof (dospath), "%sDOS_SEPE
925                 snprintf(doslink, doslink_len, "%sDOS_SEPERATOR
926                 snprintf(dospath, sizeof (dospath), "%s:%d", mai
927                 snprintf(doslink, doslink_len, "%s:%d", maindev-
928                 m = devinfo_storage_new_minor(maindev_path, dosp
929                 g_queue_push_tail (mq, m);
930             }
931         }
932         free (doslink);
933     }

934     maindev_is_d0 = (strcmp (maindev->slice, "d0") == 0);

935     /* enqueue all volumes */
936     while (!g_queue_is_empty (mq)) {
937         m = g_queue_pop_head (mq);

938         /* if main device is d0, we'll throw away s2/p0 */
939         if (maindev_is_d0 && (strcmp (m->slice, whole_disk) == 0)) {
940             devinfo_storage_free_minor (m);
941             continue;
942         }
943         /* don't do p0 on cdrom */
944         if (is_cdrom && (strcmp (m->slice, "p0") == 0)) {
945             devinfo_storage_free_minor (m);
946             continue;
947         }
948         if (rescan) {
949             /* in rescan mode, don't reprobe existing volumes */
950             /* XXX detect volume removal? */
951             volume = hal_device_store_match_key_value_string (hald_g
952             "solaris.devfs_path", m->devpath);
953             if ((volume == NULL) || !hal_device_has_capability(volum
954                 devinfo_volume_add (parent, node, m);
955             } else {
956                 HAL_INFO(("rescan volume exists %s", m->devpath)
957             }
958         } else {
959             devinfo_volume_add (parent, node, m);
960         }
961         devinfo_storage_free_minor (m);
962     }

964     if (maindev_path != NULL) {
965         di_devfs_path_free (maindev_path);
966     }

968     return;

970 err:
971     if (maindev_path != NULL) {
972         di_devfs_path_free (maindev_path);
973     }
974     if (!rescan) {
975         devinfo_add_enqueue (parent, devfs_path, &devinfo_storage_handle
976     }
977 }
_____unchanged_portion_omitted_____

1403 static gboolean
1404 is_dos_path(char *path, int *partnum)

```

```
1405 {
1406     char *p;
1408     if ((p = strrchr (path, DOS_SEPERATOR)) == NULL) {
1409         if ((p = strrchr (path, ':')) == NULL) {
1410             return (FALSE);
1411         }
1412     }
1413     return ((*partnum = atoi(p + 1)) != 0);
1414 }
```

```
1414 static gboolean
1415 dos_to_dev(char *path, char **devpath, int *partnum)
```

```
1416 {
1417     char *p;
1419     if ((p = strrchr (path, DOS_SEPERATOR)) == NULL) {
1420         if ((p = strrchr (path, ':')) == NULL) {
1421             return (FALSE);
1422         }
1423     }
1424     if ((*partnum = atoi(p + 1)) == 0) {
1425         return (FALSE);
1426     }
1427     p[0] = '\0';
1428     *devpath = strdup(path);
1429     p[0] = DOS_SEPERATOR;
1430     p[0] = ':';
1431     return (*devpath != NULL);
1432 }
```

unchanged\_portion\_omitted\_