```
**********************************************************
  910416 Fri Feb  1 17:14:32 2013
new/usr/src/uts/common/io/scsi/targets/sd.c
3515 sd gives RMW warnings for reads
Reviewed by: Albert Lee <trisk@nexenta.com>
Reviewed by: Kevin Crowe <kevin.crowe@nexenta.com>
Reviewed by: Gordon Ross <gordon.ross@nexenta.com>
**********************************************************
```

```
/*
 * 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) 1990, 2010, Oracle and/or its affiliates. All rights reserved.
 */
/*
 * Copyright 2011 Nexenta Systems, Inc.  All rights reserved.
 * Copyright (c) 2011 Bayard G. Bell.  All rights reserved.
 * Copyright (c) 2012 by Delphix. All rights reserved.
 * Copyright 2013 Nexenta Systems, Inc.  All rights reserved.
 */
/*
 * Copyright 2011 cyril.galibern@opensvc.com
 */

/*
 * SCSI disk target driver.
 */
#include <sys/scsi/scsi.h>
#include <sys/dkbad.h>
#include <sys/dklabel.h>
#include <sys/dkio.h>
#include <sys/fdio.h>
#include <sys/cdio.h>
#include <sys/mhd.h>
#include <sys/vtoc.h>
#include <sys/dktp/fdisk.h>
#include <sys/kstat.h>
#include <sys/vtrace.h>
#include <sys/note.h>
#include <sys/thread.h>
#include <sys/proc.h>
#include <sys/efi_partition.h>
#include <sys/var.h>
#include <sys/aio_req.h>

#ifdef __lock_lint
#define _LP64
#define __amd64
```



```
#endif

#if (defined(__fibre))
/* Note: is there a leadville version of the following? */
#include <sys/fc4/fcal_linkapp.h>
#endif
#include <sys/taskq.h>
#include <sys/uuid.h>
#include <sys/byteorder.h>
#include <sys/sdt.h>

#include "sd_xbuf.h"

#include <sys/scsi/targets/sddef.h>
#include <sys/cmlb.h>
#include <sys/sysevent/eventdefs.h>
#include <sys/sysevent/dev.h>

#include <sys/fm/protocol.h>

/*
 * Loadable module info.
 */
#if (defined(__fibre))
#define SD_MODULE_NAME  "SCSI SSA/FCAL Disk Driver"
#else /* !__fibre */
#define SD_MODULE_NAME  "SCSI Disk Driver"
#endif /* !__fibre */

/*
 * Define the interconnect type, to allow the driver to distinguish
 * between parallel SCSI (sd) and fibre channel (ssd) behaviors.
 *
 * This is really for backward compatibility. In the future, the driver
 * should actually check the "interconnect-type" property as reported by
 * the HBA; however at present this property is not defined by all HBAs,
 * so we will use this #define (1) to permit the driver to run in
 * backward-compatibility mode; and (2) to print a notification message
 * if an FC HBA does not support the "interconnect-type" property.  The
 * behavior of the driver will be to assume parallel SCSI behaviors unless
 * the "interconnect-type" property is defined by the HBA **AND** has a
 * value of either INTERCONNECT_FIBRE, INTERCONNECT_SSA, or
 * INTERCONNECT_FABRIC, in which case the driver will assume Fibre
 * Channel behaviors (as per the old ssd).  (Note that the
 * INTERCONNECT_1394 and INTERCONNECT_USB types are not supported and
 * will result in the driver assuming parallel SCSI behaviors.)
 *
 * (see common/sys/scsi/impl/services.h)
 *
 * Note: For ssd semantics, don't use INTERCONNECT_FABRIC as the default
 * since some FC HBAs may already support that, and there is some code in
 * the driver that already looks for it.  Using INTERCONNECT_FABRIC as the
 * default would confuse that code, and besides things should work fine
 * anyways if the FC HBA already reports INTERCONNECT_FABRIC for the
 * "interconnect_type" property.
 *
 */
#if (defined(__fibre))
#define SD_DEFAULT_INTERCONNECT_TYPE    SD_INTERCONNECT_FIBRE
#else
#define SD_DEFAULT_INTERCONNECT_TYPE    SD_INTERCONNECT_PARALLEL
#endif

/*
 * The name of the driver, established from the module name in _init.
 */
```

```
124 static  char *sd_label                 = NULL;

126 /*
127  * Driver name is unfortunately prefixed on some driver.conf properties.
128  */
129 #if (defined(__fibre))
130 #define sd_max_xfer_size                ssd_max_xfer_size
131 #define sd_config_list                  ssd_config_list
132 static  char *sd_max_xfer_size          = "ssd_max_xfer_size";
133 static  char *sd_config_list            = "ssd-config-list";
134 #else
135 static  char *sd_max_xfer_size          = "sd_max_xfer_size";
136 static  char *sd_config_list            = "sd-config-list";
137 #endif

139 /*
140  * Driver global variables
141  */

143 #if (defined(__fibre))
144 /*
145  * These #defines are to avoid namespace collisions that occur because this
146  * code is currently used to compile two separate driver modules: sd and ssd.
147  * All global variables need to be treated this way (even if declared static)
148  * in order to allow the debugger to resolve the names properly.
149  * It is anticipated that in the near future the ssd module will be obsoleted,
150  * at which time this namespace issue should go away.
151  */
152 #define sd_state                        ssd_state
153 #define sd_io_time                      ssd_io_time
154 #define sd_failfast_enable              ssd_failfast_enable
155 #define sd_ua_retry_count               ssd_ua_retry_count
156 #define sd_report_pfa                   ssd_report_pfa
157 #define sd_max_throttle                 ssd_max_throttle
158 #define sd_min_throttle                 ssd_min_throttle
159 #define sd_rot_delay                    ssd_rot_delay

161 #define sd_retry_on_reservation_conflict        \
162                                         ssd_retry_on_reservation_conflict
163 #define sd_reinstate_resv_delay         ssd_reinstate_resv_delay
164 #define sd_resv_conflict_name           ssd_resv_conflict_name

166 #define sd_component_mask               ssd_component_mask
167 #define sd_level_mask                   ssd_level_mask
168 #define sd_debug_un                     ssd_debug_un
169 #define sd_error_level                  ssd_error_level

171 #define sd_xbuf_active_limit            ssd_xbuf_active_limit
172 #define sd_xbuf_reserve_limit           ssd_xbuf_reserve_limit

174 #define sd_tr                           ssd_tr
175 #define sd_reset_throttle_timeout       ssd_reset_throttle_timeout
176 #define sd_qfull_throttle_timeout       ssd_qfull_throttle_timeout
177 #define sd_qfull_throttle_enable        ssd_qfull_throttle_enable
178 #define sd_check_media_time             ssd_check_media_time
179 #define sd_wait_cmds_complete           ssd_wait_cmds_complete
180 #define sd_label_mutex                  ssd_label_mutex
181 #define sd_detach_mutex                 ssd_detach_mutex
182 #define sd_log_buf                      ssd_log_buf
183 #define sd_log_mutex                    ssd_log_mutex

185 #define sd_disk_table                   ssd_disk_table
186 #define sd_disk_table_size              ssd_disk_table_size
187 #define sd_sense_mutex                  ssd_sense_mutex
188 #define sd_cdbtab                       ssd_cdbtab
```



```
190 #define sd_cb_ops                       ssd_cb_ops
191 #define sd_ops                          ssd_ops
192 #define sd_additional_codes             ssd_additional_codes
193 #define sd_tgops                        ssd_tgops

195 #define sd_minor_data                   ssd_minor_data
196 #define sd_minor_data_efi               ssd_minor_data_efi

198 #define sd_tq                           ssd_tq
199 #define sd_wmr_tq                       ssd_wmr_tq
200 #define sd_taskq_name                   ssd_taskq_name
201 #define sd_wmr_taskq_name               ssd_wmr_taskq_name
202 #define sd_taskq_minalloc               ssd_taskq_minalloc
203 #define sd_taskq_maxalloc               ssd_taskq_maxalloc

205 #define sd_dump_format_string           ssd_dump_format_string

207 #define sd_iostart_chain                ssd_iostart_chain
208 #define sd_iodone_chain                 ssd_iodone_chain

210 #define sd_pm_idletime                  ssd_pm_idletime

212 #define sd_force_pm_supported           ssd_force_pm_supported

214 #define sd_dtype_optical_bind           ssd_dtype_optical_bind

216 #define sd_ssc_init                     ssd_ssc_init
217 #define sd_ssc_send                     ssd_ssc_send
218 #define sd_ssc_fini                     ssd_ssc_fini
219 #define sd_ssc_assessment               ssd_ssc_assessment
220 #define sd_ssc_post                     ssd_ssc_post
221 #define sd_ssc_print                    ssd_ssc_print
222 #define sd_ssc_ereport_post             ssd_ssc_ereport_post
223 #define sd_ssc_set_info                 ssd_ssc_set_info
224 #define sd_ssc_extract_info             ssd_ssc_extract_info

226 #endif

228 #ifdef  SDDEBUG
229 int     sd_force_pm_supported           = 0;
230 #endif  /* SDDEBUG */

232 void *sd_state                          = NULL;
233 int sd_io_time                          = SD_IO_TIME;
234 int sd_failfast_enable                  = 1;
235 int sd_ua_retry_count                   = SD_UA_RETRY_COUNT;
236 int sd_report_pfa                       = 1;
237 int sd_max_throttle                     = SD_MAX_THROTTLE;
238 int sd_min_throttle                     = SD_MIN_THROTTLE;
239 int sd_rot_delay                        = 4; /* Default 4ms Rotation delay */
240 int sd_qfull_throttle_enable            = TRUE;

242 int sd_retry_on_reservation_conflict    = 1;
243 int sd_reinstate_resv_delay             = SD_REINSTATE_RESV_DELAY;
244 _NOTE(SCHEME_PROTECTS_DATA("safe sharing", sd_reinstate_resv_delay))

246 static int sd_dtype_optical_bind        = -1;

248 /* Note: the following is not a bug, it really is "sd_" and not "ssd_" */
249 static  char *sd_resv_conflict_name     = "sd_retry_on_reservation_conflict";

251 /*
252  * Global data for debug logging. To enable debug printing, sd_component_mask
253  * and sd_level_mask should be set to the desired bit patterns as outlined in
254  * sddef.h.
255  */
```

```
 256 uint_t  sd_component_mask               = 0x0;
 257 uint_t  sd_level_mask                  = 0x0;
 258 struct  sd_lun *sd_debug_un             = NULL;
 259 uint_t  sd_error_level                 = SCSI_ERR_RETRYABLE;

 261 /* Note: these may go away in the future... */
 262 static uint32_t sd_xbuf_active_limit    = 512;
 263 static uint32_t sd_xbuf_reserve_limit   = 16;

 265 static struct sd_resv_reclaim_request   sd_tr = { NULL, NULL, NULL, 0, 0, 0 };

 267 /*
 268  * Timer value used to reset the throttle after it has been reduced
 269  * (typically in response to TRAN_BUSY or STATUS_QFULL)
 270  */
 271 static int sd_reset_throttle_timeout    = SD_RESET_THROTTLE_TIMEOUT;
 272 static int sd_qfull_throttle_timeout    = SD_QFULL_THROTTLE_TIMEOUT;

 274 /*
 275  * Interval value associated with the media change scsi watch.
 276  */
 277 static int sd_check_media_time          = 3000000;

 279 /*
 280  * Wait value used for in progress operations during a DDI_SUSPEND
 281  */
 282 static int sd_wait_cmds_complete        = SD_WAIT_CMDS_COMPLETE;

 284 /*
 285  * sd_label_mutex protects a static buffer used in the disk label
 286  * component of the driver
 287  */
 288 static kmutex_t sd_label_mutex;

 290 /*
 291  * sd_detach_mutex protects un_layer_count, un_detach_count, and
 292  * un_opens_in_progress in the sd_lun structure.
 293  */
 294 static kmutex_t sd_detach_mutex;

 296 _NOTE(MUTEX_PROTECTS_DATA(sd_detach_mutex,
 297         sd_lun::{un_layer_count un_detach_count un_opens_in_progress}))

 299 /*
 300  * Global buffer and mutex for debug logging
 301  */
 302 static char     sd_log_buf[1024];
 303 static kmutex_t sd_log_mutex;

 305 /*
 306  * Structs and globals for recording attached lun information.
 307  * This maintains a chain. Each node in the chain represents a SCSI controller.
 308  * The structure records the number of luns attached to each target connected
 309  * with the controller.
 310  * For parallel scsi device only.
 311  */
 312 struct sd_scsi_hba_tgt_lun {
 313         struct sd_scsi_hba_tgt_lun      *next;
 314         dev_info_t                      *pdip;
 315         int                             nlun[NTARGETS_WIDE];
 316 };
_____unchanged_portion_omitted_


12528 /*
12529  *    Function: sd_mapblockaddr_iostart
```



```
12530  *
12531  * Description: Verify request lies within the partition limits for
12532  *              the indicated minor device.  Issue "overrun" buf if
12533  *              request would exceed partition range.  Converts
12534  *              partition-relative block address to absolute.
12535  *
12536  *              Upon exit of this function:
12537  *              1.I/O is aligned
12538  *                 xp->xb_blkno represents the absolute sector address
12539  *              2.I/O is misaligned
12540  *                 xp->xb_blkno represents the absolute logical block address
12541  *                 based on DEV_BSIZE. The logical block address will be
12542  *                 converted to physical sector address in sd_mapblocksize_\
12543  *                 iostart.
12544  *              3.I/O is misaligned but is aligned in "overrun" buf
12545  *                 xp->xb_blkno represents the absolute logical block address
12546  *                 based on DEV_BSIZE. The logical block address will be
12547  *                 converted to physical sector address in sd_mapblocksize_\
12548  *                 iostart. But no RMW will be issued in this case.
12549  *
12550  *     Context: Can sleep
12551  *
12552  *      Issues: This follows what the old code did, in terms of accessing
12553  *              some of the partition info in the unit struct without holding
12554  *              the mutext.  This is a general issue, if the partition info
12555  *              can be altered while IO is in progress... as soon as we send
12556  *              a buf, its partitioning can be invalid before it gets to the
12557  *              device.  Probably the right fix is to move partitioning out
12558  *              of the driver entirely.
12559  */

12561 static void
12562 sd_mapblockaddr_iostart(int index, struct sd_lun *un, struct buf *bp)
12563 {
12564         diskaddr_t      nblocks;        /* #blocks in the given partition */
12565         daddr_t blocknum;       /* Block number specified by the buf */
12566         size_t  requested_nblocks;
12567         size_t  available_nblocks;
12568         int     partition;
12569         diskaddr_t      partition_offset;
12570         struct sd_xbuf *xp;
12571         int secmask = 0, blknomask = 0;
12572         ushort_t is_aligned = TRUE;

12574         ASSERT(un != NULL);
12575         ASSERT(bp != NULL);
12576         ASSERT(!mutex_owned(SD_MUTEX(un)));

12578         SD_TRACE(SD_LOG_IO_PARTITION, un,
12579             "sd_mapblockaddr_iostart: entry: buf:0x%p\n", bp);

12581         xp = SD_GET_XBUF(bp);
12582         ASSERT(xp != NULL);

12584         /*
12585          * If the geometry is not indicated as valid, attempt to access
12586          * the unit & verify the geometry/label. This can be the case for
12587          * removable-media devices, of if the device was opened in
12588          * NDELAY/NONBLOCK mode.
12589          */
12590         partition = SDPART(bp->b_edev);

12592         if (!SD_IS_VALID_LABEL(un)) {
12593                 sd_ssc_t *ssc;
12594                 /*
12595                  * Initialize sd_ssc_t for internal uscsi commands
```

```
12596                  * In case of potential performance issue, we need
12597                  * to alloc memory only if there is invalid label
12598                  */
12599                 ssc = sd_ssc_init(un);

12601                 if (sd_ready_and_valid(ssc, partition) != SD_READY_VALID) {
12602                         /*
12603                          * For removable devices it is possible to start an
12604                          * I/O without a media by opening the device in nodelay
12605                          * mode. Also for writable CDs there can be many
12606                          * scenarios where there is no geometry yet but volume
12607                          * manager is trying to issue a read() just because
12608                          * it can see TOC on the CD. So do not print a message
12609                          * for removables.
12610                          */
12611                         if (!un->un_f_has_removable_media) {
12612                                 scsi_log(SD_DEVINFO(un), sd_label, CE_WARN,
12613                                     "i/o to invalid geometry\n");
12614                         }
12615                         bioerror(bp, EIO);
12616                         bp->b_resid = bp->b_bcount;
12617                         SD_BEGIN_IODONE(index, un, bp);

12619                         sd_ssc_fini(ssc);
12620                         return;
12621                 }
12622                 sd_ssc_fini(ssc);
12623         }

12625         nblocks = 0;
12626         (void) cmlb_partinfo(un->un_cmlbhandle, partition,
12627             &nblocks, &partition_offset, NULL, NULL, (void *)SD_PATH_DIRECT);

12629         if (un->un_f_enable_rmw) {
12630                 blknomask = (un->un_phy_blocksize / DEV_BSIZE) - 1;
12631                 secmask = un->un_phy_blocksize - 1;
12632         } else {
12633                 blknomask = (un->un_tgt_blocksize / DEV_BSIZE) - 1;
12634                 secmask = un->un_tgt_blocksize - 1;
12635         }

12637         if ((bp->b_lblkno & (blknomask)) || (bp->b_bcount & (secmask))) {
12638                 is_aligned = FALSE;
12639         }

12641         if (!(NOT_DEVBSIZE(un)) || un->un_f_enable_rmw) {
12642                 /*
12643                  * If I/O is aligned, no need to involve RMW(Read Modify Write)
12644                  * Convert the logical block number to target's physical sector
12645                  * number.
12646                  */
12647                 if (is_aligned) {
12648                         xp->xb_blkno = SD_SYS2TGTBLOCK(un, xp->xb_blkno);
12649                 } else {
12650                         /*
12651                          * There is no RMW if we're just reading, so don't
12652                          * warn or error out because of it.
12653                          */
12654                         if (bp->b_flags & B_READ) {
12655                                 /*EMPTY*/
12656                         } else if (!un->un_f_enable_rmw &&
12657                             un->un_f_rmw_type == SD_RMW_TYPE_RETURN_ERROR) {
12650                         switch (un->un_f_rmw_type) {
12651                         case SD_RMW_TYPE_RETURN_ERROR:
12652                                 if (un->un_f_enable_rmw)
12653                                         break;
```



```
12654                                 else {
12658                                 bp->b_flags |= B_ERROR;
12659                                 goto error_exit;
12660                         } else if (un->un_f_rmw_type == SD_RMW_TYPE_DEFAULT) {
12657                         }

12659                         case SD_RMW_TYPE_DEFAULT:
12661                                 mutex_enter(SD_MUTEX(un));
12662                                 if (!un->un_f_enable_rmw &&
12663                                     un->un_rmw_msg_timeid == NULL) {
12664                                         scsi_log(SD_DEVINFO(un), sd_label,
12665                                             CE_WARN, "I/O request is not "
12666                                             "aligned with %d disk sector size. "
12667                                             "It is handled through Read Modify "
12668                                             "Write but the performance is "
12669                                             "very low.\n",
12670                                             un->un_tgt_blocksize);
12671                                         un->un_rmw_msg_timeid =
12672                                             timeout(sd_rmw_msg_print_handler,
12673                                             un, SD_RMW_MSG_PRINT_TIMEOUT);
12674                                 } else {
12675                                         un->un_rmw_incre_count ++;
12676                                 }
12677                                 mutex_exit(SD_MUTEX(un));
12677                                 break;

12679                         case SD_RMW_TYPE_NO_WARNING:
12680                         default:
12681                                 break;
12678                         }

12680                         nblocks = SD_TGT2SYSBLOCK(un, nblocks);
12681                         partition_offset = SD_TGT2SYSBLOCK(un,
12682                             partition_offset);
12683                 }
12684         }

12686         /*
12687          * blocknum is the starting block number of the request. At this
12688          * point it is still relative to the start of the minor device.
12689          */
12690         blocknum = xp->xb_blkno;

12692         /*
12693          * Legacy: If the starting block number is one past the last block
12694          * in the partition, do not set B_ERROR in the buf.
12695          */
12696         if (blocknum == nblocks)  {
12697                 goto error_exit;
12698         }

12700         /*
12701          * Confirm that the first block of the request lies within the
12702          * partition limits. Also the requested number of bytes must be
12703          * a multiple of the system block size.
12704          */
12705         if ((blocknum < 0) || (blocknum >= nblocks) ||
12706             ((bp->b_bcount & (DEV_BSIZE - 1)) != 0)) {
12707                 bp->b_flags |= B_ERROR;
12708                 goto error_exit;
12709         }

12711         /*
12712          * If the requsted # blocks exceeds the available # blocks, that
12713          * is an overrun of the partition.
12714          */
```

```
12715         if ((!NOT_DEVBSIZE(un)) && is_aligned) {
12716                 requested_nblocks = SD_BYTES2TGTBLOCKS(un, bp->b_bcount);
12717         } else {
12718                 requested_nblocks = SD_BYTES2SYSBLOCKS(bp->b_bcount);
12719         }

12721         available_nblocks = (size_t)(nblocks - blocknum);
12722         ASSERT(nblocks >= blocknum);

12724         if (requested_nblocks > available_nblocks) {
12725                 size_t resid;

12727                 /*
12728                  * Allocate an "overrun" buf to allow the request to proceed
12729                  * for the amount of space available in the partition. The
12730                  * amount not transferred will be added into the b_resid
12731                  * when the operation is complete. The overrun buf
12732                  * replaces the original buf here, and the original buf
12733                  * is saved inside the overrun buf, for later use.
12734                  */
12735                 if ((!NOT_DEVBSIZE(un)) && is_aligned) {
12736                         resid = SD_TGTBLOCKS2BYTES(un,
12737                             (offset_t)(requested_nblocks - available_nblocks));
12738                 } else {
12739                         resid = SD_SYSBLOCKS2BYTES(
12740                             (offset_t)(requested_nblocks - available_nblocks));
12741                 }

12743                 size_t count = bp->b_bcount - resid;
12744                 /*
12745                  * Note: count is an unsigned entity thus it'll NEVER
12746                  * be less than 0 so ASSERT the original values are
12747                  * correct.
12748                  */
12749                 ASSERT(bp->b_bcount >= resid);

12751                 bp = sd_bioclone_alloc(bp, count, blocknum,
12752                     (int (*)(struct buf *)) sd_mapblockaddr_iodone);
12753                 xp = SD_GET_XBUF(bp); /* Update for 'new' bp! */
12754                 ASSERT(xp != NULL);
12755         }

12757         /* At this point there should be no residual for this buf. */
12758         ASSERT(bp->b_resid == 0);

12760         /* Convert the block number to an absolute address. */
12761         xp->xb_blkno += partition_offset;

12763         SD_NEXT_IOSTART(index, un, bp);

12765         SD_TRACE(SD_LOG_IO_PARTITION, un,
12766             "sd_mapblockaddr_iostart: exit 0: buf:0x%p\n", bp);

12768         return;

12770 error_exit:
12771         bp->b_resid = bp->b_bcount;
12772         SD_BEGIN_IODONE(index, un, bp);
12773         SD_TRACE(SD_LOG_IO_PARTITION, un,
12774             "sd_mapblockaddr_iostart: exit 1: buf:0x%p\n", bp);
12775 }
```

_____***unchanged_portion_omitted_***