96

```
/*
 *   Copyright (c) International Business Machines Corp., 2000−2002
 *
 *   This program is free software;  you can redistribute it and/or modify
 *   it under the terms of the GNU General Public License as published by
 *   the Free Software Foundation; either version 2 of the License, or
 *   (at your option) any later version.
 *
 *   This program is distributed in the hope that it will be useful,
 *   but WITHOUT ANY WARRANTY;  without even the implied warranty of
 *   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See
 *   the GNU General Public License for more details.
 *
 *   You should have received a copy of the GNU General Public License
 *   along with this program;  if not, write to the Free Software
 *   Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111−1307 USA
 */

#include <linux/fs.h>
#include "jfs_incore.h"
#include "jfs_dmap.h"
#include "jfs_imap.h"
#include "jfs_lock.h"
#include "jfs_metapage.h"
#include "jfs_debug.h"

/*
 *      Debug code for double−checking block map
 */
/* #define      _JFS_DEBUG_DMAP 1 */

#ifdef  _JFS_DEBUG_DMAP
#define DBINITMAP(size,ipbmap,results) \
        DBinitmap(size,ipbmap,results)
#define DBALLOC(dbmap,mapsize,blkno,nblocks) \
        DBAlloc(dbmap,mapsize,blkno,nblocks)
#define DBFREE(dbmap,mapsize,blkno,nblocks) \
        DBFree(dbmap,mapsize,blkno,nblocks)
#define DBALLOCCK(dbmap,mapsize,blkno,nblocks) \
        DBAllocCK(dbmap,mapsize,blkno,nblocks)
#define DBFREECK(dbmap,mapsize,blkno,nblocks) \
        DBFreeCK(dbmap,mapsize,blkno,nblocks)

static void DBinitmap(s64, struct inode *, u32 **);
static void DBAlloc(uint *, s64, s64, s64);
static void DBFree(uint *, s64, s64, s64);
static void DBAllocCK(uint *, s64, s64, s64);
static void DBFreeCK(uint *, s64, s64, s64);
#else
#define DBINITMAP(size,ipbmap,results)
#define DBALLOC(dbmap, mapsize, blkno, nblocks)
#define DBFREE(dbmap, mapsize, blkno, nblocks)
#define DBALLOCCK(dbmap, mapsize, blkno, nblocks)
#define DBFREECK(dbmap, mapsize, blkno, nblocks)
#endif                          /* _JFS_DEBUG_DMAP */

/*
 *      SERIALIZATION of the Block Allocation Map.
 *
 *      the working state of the block allocation map is accessed in
 *      two directions:
 *
 *      1) allocation and free requests that start at the dmap
 *         level and move up through the dmap control pages (i.e.
 *         the vast majority of requests).
 *
 *      2) allocation requests that start at dmap control page
 *         level and work down towards the dmaps.
 *
 *      the serialization scheme used here is as follows.
 *
 *      requests which start at the bottom are serialized against each
 *      other through buffers and each requests holds onto its buffers
 *      as it works it way up from a single dmap to the required level
 *      of dmap control page.
 *      requests that start at the top are serialized against each other
 *      and request that start from the bottom by the multiple read/single
 *      write inode lock of the bmap inode. requests starting at the top
 *      take this lock in write mode while request starting at the bottom
 *      take the lock in read mode.  a single top−down request may proceed
 *      exclusively while multiple bottoms−up requests may proceed
 *      simultaneously (under the protection of busy buffers).
 *
 *      in addition to information found in dmaps and dmap control pages,
 *      the working state of the block allocation map also includes read/
 *      write information maintained in the bmap descriptor (i.e. total
 *      free block count, allocation group level free block counts).
 *      a single exclusive lock (BMAP_LOCK) is used to guard this information
 *      in the face of multiple−bottoms up requests.
 *      (lock ordering: IREAD_LOCK, BMAP_LOCK);
```

```
 *
 *      accesses to the persistent state of the block allocation map (limited
 *      to the persistent bitmaps in dmaps) is guarded by (busy) buffers.
 */

#define BMAP_LOCK_INIT(bmp)     init_MUTEX(&bmp->db_bmaplock)
#define BMAP_LOCK(bmp)          down(&bmp->db_bmaplock)
#define BMAP_UNLOCK(bmp)        up(&bmp->db_bmaplock)

/*
 * forward references
 */
static void dbAllocBits(struct bmap * bmp, struct dmap * dp, s64 blkno,
                        int nblocks);
static void dbSplit(dmtree_t * tp, int leafno, int splitsz, int newval);
static void dbBackSplit(dmtree_t * tp, int leafno);
static void dbJoin(dmtree_t * tp, int leafno, int newval);
static void dbAdjTree(dmtree_t * tp, int leafno, int newval);
static int dbAdjCtl(struct bmap * bmp, s64 blkno, int newval, int alloc,
                    int level);
static int dbAllocAny(struct bmap * bmp, s64 nblocks, int l2nb, s64 * results);
static int dbAllocNext(struct bmap * bmp, struct dmap * dp, s64 blkno,
                       int nblocks);
static int dbAllocNear(struct bmap * bmp, struct dmap * dp, s64 blkno,
                       int nblocks,
                       int l2nb, s64 * results);
static int dbAllocDmap(struct bmap * bmp, struct dmap * dp, s64 blkno,
                       int nblocks);
static int dbAllocDmapLev(struct bmap * bmp, struct dmap * dp, int nblocks,
                          int l2nb,
                          s64 * results);
static int dbAllocAG(struct bmap * bmp, int agno, s64 nblocks, int l2nb,
                     s64 * results);
static int dbAllocCtl(struct bmap * bmp, s64 nblocks, int l2nb, s64 blkno,
                      s64 * results);
int dbExtend(struct inode *ip, s64 blkno, s64 nblocks, s64 addnblocks);
static int dbFindBits(u32 word, int l2nb);
static int dbFindCtl(struct bmap * bmp, int l2nb, int level, s64 * blkno);
static int dbFindLeaf(dmtree_t * tp, int l2nb, int *leafidx);
static void dbFreeBits(struct bmap * bmp, struct dmap * dp, s64 blkno,
                       int nblocks);
static int dbFreeDmap(struct bmap * bmp, struct dmap * dp, s64 blkno,
                      int nblocks);
static int dbMaxBud(u8 * cp);
s64 dbMapFileSizeToMapSize(struct inode *ipbmap);
int blkstol2(s64 nb);
void fsDirty(void);

int cntlz(u32 value);
int cnttz(u32 word);

static int dbAllocDmapBU(struct bmap * bmp, struct dmap * dp, s64 blkno,
                         int nblocks);
static int dbInitDmap(struct dmap * dp, s64 blkno, int nblocks);
static int dbInitDmapTree(struct dmap * dp);
static int dbInitTree(struct dmaptree * dtp);
static int dbInitDmapCtl(struct dmapctl * dcp, int level, int i);
static int dbGetL2AGSize(s64 nblocks);

/*
 *      buddy table
 *
 * table used for determining buddy sizes within characters of
 * dmap bitmap words.  the characters themselves serve as indexes
 * into the table, with the table elements yielding the maximum
 * binary buddy of free bits within the character.
 */
signed char budtab[256] = {
        3, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
        2, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
        2, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
        2, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
        2, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
        2, 1, 1, 1, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0,
        2, 1, 1, 1, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0,
        2, 1, 1, 1, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0,
        2, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
        2, 1, 1, 1, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0,
        2, 1, 1, 1, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0,
        2, 1, 1, 1, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0,
        2, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
        2, 1, 1, 1, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0,
        2, 1, 1, 1, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0,
        2, 1, 1, 1, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, -1
};


/*
 * NAME:        dbMount()
 *
```

```
 * FUNCTION:    initialize the block allocation map.
 *
 *              memory is allocated for the in−core bmap descriptor and
 *              the in−core descriptor is initialized from disk.
 *
 * PARAMETERS:
 *      ipbmap  −  pointer to in−core inode for the block map.
 *
 * RETURN VALUES:
 *      0       − success
 *      ENOMEM  − insufficient memory
 *      EIO     − i/o error
 */
int dbMount(struct inode *ipbmap)
{
        struct bmap *bmp;
        struct dbmap *dbmp_le;
        struct metapage *mp;
        int i;

        /*
         * allocate/initialize the in−memory bmap descriptor
         */
        /* allocate memory for the in−memory bmap descriptor */
        bmp = kmalloc(sizeof(struct bmap), GFP_KERNEL);
        if (bmp == NULL)
                return (ENOMEM);

        /* read the on−disk bmap descriptor. */
        mp = read_metapage(ipbmap,
                           BMAPBLKNO << JFS_SBI(ipbmap−>i_sb)−>l2nbperpage,
                           PSIZE, 0);
        if (mp == NULL) {
                kfree(bmp);
                return (EIO);
        }

        /* copy the on−disk bmap descriptor to its in−memory version. */
        dbmp_le = (struct dbmap *) mp−>data;
        bmp−>db_mapsize = le64_to_cpu(dbmp_le−>dn_mapsize);
        bmp−>db_nfree = le64_to_cpu(dbmp_le−>dn_nfree);
        bmp−>db_l2nbperpage = le32_to_cpu(dbmp_le−>dn_l2nbperpage);
        bmp−>db_numag = le32_to_cpu(dbmp_le−>dn_numag);
        bmp−>db_maxlevel = le32_to_cpu(dbmp_le−>dn_maxlevel);
        bmp−>db_maxag = le32_to_cpu(dbmp_le−>dn_maxag);
        bmp−>db_agpref = le32_to_cpu(dbmp_le−>dn_agpref);
        bmp−>db_aglevel = le32_to_cpu(dbmp_le−>dn_aglevel);
        bmp−>db_agheigth = le32_to_cpu(dbmp_le−>dn_agheigth);
        bmp−>db_agwidth = le32_to_cpu(dbmp_le−>dn_agwidth);
        bmp−>db_agstart = le32_to_cpu(dbmp_le−>dn_agstart);
        bmp−>db_agl2size = le32_to_cpu(dbmp_le−>dn_agl2size);
        for (i = 0; i < MAXAG; i++)
                bmp−>db_agfree[i] = le64_to_cpu(dbmp_le−>dn_agfree[i]);
        bmp−>db_agsize = le64_to_cpu(dbmp_le−>dn_agsize);
        bmp−>db_maxfreebud = dbmp_le−>dn_maxfreebud;

        /* release the buffer. */
        release_metapage(mp);

        /* bind the bmap inode and the bmap descriptor to each other. */
        bmp−>db_ipbmap = ipbmap;
        JFS_SBI(ipbmap−>i_sb)−>bmap = bmp;

        memset(bmp−>db_active, 0, sizeof(bmp−>db_active));
        DBINITMAP(bmp−>db_mapsize, ipbmap, &bmp−>db_DBmap);

        /*
         * allocate/initialize the bmap lock
         */
        BMAP_LOCK_INIT(bmp);

        return (0);
}


/*
 * NAME:        dbUnmount()
 *
 * FUNCTION:    terminate the block allocation map in preparation for
 *              file system unmount.
 *
 *              the in−core bmap descriptor is written to disk and
 *              the memory for this descriptor is freed.
 *
 * PARAMETERS:
 *      ipbmap  −  pointer to in−core inode for the block map.
 *
 * RETURN VALUES:
 *      0       − success
 *      EIO     − i/o error
```

```
 */
int dbUnmount(struct inode *ipbmap, int mounterror)
{
        struct bmap *bmp = JFS_SBI(ipbmap->i_sb)->bmap;
        int i;

        if (!(mounterror || isReadOnly(ipbmap)))
                dbSync(ipbmap);

        /*
         * Invalidate the page cache buffers
         */
        truncate_inode_pages(ipbmap->i_mapping, 0);

        /*
         * Sanity Check
         */
        for (i = 0; i < bmp->db_numag; i++)
                if (atomic_read(&bmp->db_active[i]))
                        printk(KERN_ERR "dbUnmount: db_active[%d] = %d\n",
                               i, atomic_read(&bmp->db_active[i]));

        /* free the memory for the in-memory bmap. */
        kfree(bmp);

        return (0);
}

/*
 *      dbSync()
 */
int dbSync(struct inode *ipbmap)
{
        struct dbmap *dbmp_le;
        struct bmap *bmp = JFS_SBI(ipbmap->i_sb)->bmap;
        struct metapage *mp;
        int i;

        /*
         * write bmap global control page
         */
        /* get the buffer for the on-disk bmap descriptor. */
        mp = read_metapage(ipbmap,
                           BMAPBLKNO << JFS_SBI(ipbmap->i_sb)->l2nbperpage,
                           PSIZE, 0);
        if (mp == NULL) {
                jERROR(1,("dbSync: read_metapage failed!\n"));
                return (EIO);
        }
        /* copy the in-memory version of the bmap to the on-disk version */
        dbmp_le = (struct dbmap *) mp->data;
        dbmp_le->dn_mapsize = cpu_to_le64(bmp->db_mapsize);
        dbmp_le->dn_nfree = cpu_to_le64(bmp->db_nfree);
        dbmp_le->dn_l2nbperpage = cpu_to_le32(bmp->db_l2nbperpage);
        dbmp_le->dn_numag = cpu_to_le32(bmp->db_numag);
        dbmp_le->dn_maxlevel = cpu_to_le32(bmp->db_maxlevel);
        dbmp_le->dn_maxag = cpu_to_le32(bmp->db_maxag);
        dbmp_le->dn_agpref = cpu_to_le32(bmp->db_agpref);
        dbmp_le->dn_aglevel = cpu_to_le32(bmp->db_aglevel);
        dbmp_le->dn_agheigth = cpu_to_le32(bmp->db_agheigth);
        dbmp_le->dn_agwidth = cpu_to_le32(bmp->db_agwidth);
        dbmp_le->dn_agstart = cpu_to_le32(bmp->db_agstart);
        dbmp_le->dn_agl2size = cpu_to_le32(bmp->db_agl2size);
        for (i = 0; i < MAXAG; i++)
                dbmp_le->dn_agfree[i] = cpu_to_le64(bmp->db_agfree[i]);
        dbmp_le->dn_agsize = cpu_to_le64(bmp->db_agsize);
        dbmp_le->dn_maxfreebud = bmp->db_maxfreebud;

        /* write the buffer */
        write_metapage(mp);

        /*
         * write out dirty pages of bmap
         */
        fsync_inode_data_buffers(ipbmap);

        ipbmap->i_state |= I_DIRTY;
        diWriteSpecial(ipbmap, 0);

        return (0);
}


/*
 * NAME:        dbFree()
 *
 * FUNCTION:    free the specified block range from the working block
 *              allocation map.
 *
 *              the blocks will be free from the working map one dmap
```

```
361   *              at a time.
362   *
363   * PARAMETERS:
364   *      ip      −  pointer to in−core inode;
365   *      blkno   −  starting block number to be freed.
366   *      nblocks −  number of blocks to be freed.
367   *
368   * RETURN VALUES:
369   *      0       − success
370   *      EIO     − i/o error
371   */
372  int dbFree(struct inode *ip, s64 blkno, s64 nblocks)
373  {
374          struct metapage *mp;
375          struct dmap *dp;
376          int nb, rc;
377          s64 lblkno, rem;
378          struct inode *ipbmap = JFS_SBI(ip−>i_sb)−>ipbmap;
379          struct bmap *bmp = JFS_SBI(ip−>i_sb)−>bmap;
380
381          IREAD_LOCK(ipbmap);
382
383          /* block to be freed better be within the mapsize. */
384          assert(blkno + nblocks <= bmp−>db_mapsize);
385
386          /*
387           * free the blocks a dmap at a time.
388           */
389          mp = NULL;
390          for (rem = nblocks; rem > 0; rem −= nb, blkno += nb) {
391                  /* release previous dmap if any */
392                  if (mp) {
393                          write_metapage(mp);
394                  }
395
396                  /* get the buffer for the current dmap. */
397                  lblkno = BLKTODMAP(blkno, bmp−>db_l2nbperpage);
398                  mp = read_metapage(ipbmap, lblkno, PSIZE, 0);
399                  if (mp == NULL) {
400                          IREAD_UNLOCK(ipbmap);
401                          return (EIO);
402                  }
403                  dp = (struct dmap *) mp−>data;
404
405                  /* determine the number of blocks to be freed from
406                   * this dmap.
407                   */
408                  nb = min(rem, BPERDMAP − (blkno & (BPERDMAP − 1)));
409
410                  DBALLOCCK(bmp−>db_DBmap, bmp−>db_mapsize, blkno, nb);
411
412                  /* free the blocks. */
413                  if ((rc = dbFreeDmap(bmp, dp, blkno, nb))) {
414                          release_metapage(mp);
415                          IREAD_UNLOCK(ipbmap);
416                          return (rc);
417                  }
418
419                  DBFREE(bmp−>db_DBmap, bmp−>db_mapsize, blkno, nb);
420          }
421
422          /* write the last buffer. */
423          write_metapage(mp);
424
425          IREAD_UNLOCK(ipbmap);
426
427          return (0);
428  }
429
430
431  /*
432   * NAME:        dbUpdatePMap()
433   *
434   * FUNCTION:    update the allocation state (free or allocate) of the
435   *              specified block range in the persistent block allocation map.
436   *
437   *              the blocks will be updated in the persistent map one
438   *              dmap at a time.
439   *
440   * PARAMETERS:
441   *      ipbmap  −  pointer to in−core inode for the block map.
442   *      free    − TRUE if block range is to be freed from the persistent
443   *                map; FALSE if it is to   be allocated.
444   *      blkno   −  starting block number of the range.
445   *      nblocks −  number of contiguous blocks in the range.
446   *      tblk    −  transaction block;
447   *
448   * RETURN VALUES:
449   *      0       − success
450   *      EIO     − i/o error
```

```
 */
int
dbUpdatePMap(struct inode *ipbmap,
             int free, s64 blkno, s64 nblocks, struct tblock * tblk)
{
        int nblks, dbitno, wbitno, rbits;
        int word, nbits, nwords;
        struct bmap *bmp = JFS_SBI(ipbmap->i_sb)->bmap;
        s64 lblkno, rem, lastlblkno;
        u32 mask;
        struct dmap *dp;
        struct metapage *mp;
        struct jfs_log *log;
        int lsn, difft, diffp;

        /* the blocks better be within the mapsize. */
        assert(blkno + nblocks <= bmp->db_mapsize);

        /* compute delta of transaction lsn from log syncpt */
        lsn = tblk->lsn;
        log = (struct jfs_log *) JFS_SBI(tblk->sb)->log;
        logdiff(difft, lsn, log);

        /*
         * update the block state a dmap at a time.
         */
        mp = NULL;
        lastlblkno = 0;
        for (rem = nblocks; rem > 0; rem -= nblks, blkno += nblks) {
                /* get the buffer for the current dmap. */
                lblkno = BLKTODMAP(blkno, bmp->db_l2nbperpage);
                if (lblkno != lastlblkno) {
                        if (mp) {
                                write_metapage(mp);
                        }

                        mp = read_metapage(bmp->db_ipbmap, lblkno, PSIZE,
                                           0);
                        if (mp == NULL)
                                return (EIO);
                }
                dp = (struct dmap *) mp->data;

                /* determine the bit number and word within the dmap of
                 * the starting block.  also determine how many blocks
                 * are to be updated within this dmap.
                 */
                dbitno = blkno & (BPERDMAP - 1);
                word = dbitno >> L2DBWORD;
                nblks = min(rem, (s64)BPERDMAP - dbitno);

                /* update the bits of the dmap words. the first and last
                 * words may only have a subset of their bits updated. if
                 * this is the case, we'll work against that word (i.e.
                 * partial first and/or last) only in a single pass.  a
                 * single pass will also be used to update all words that
                 * are to have all their bits updated.
                 */
                for (rbits = nblks; rbits > 0;
                     rbits -= nbits, dbitno += nbits) {
                        /* determine the bit number within the word and
                         * the number of bits within the word.
                         */
                        wbitno = dbitno & (DBWORD - 1);
                        nbits = min(rbits, DBWORD - wbitno);

                        /* check if only part of the word is to be updated. */
                        if (nbits < DBWORD) {
                                /* update (free or allocate) the bits
                                 * in this word.
                                 */
                                mask =
                                    (ONES << (DBWORD - nbits) >> wbitno);
                                if (free)
                                        dp->pmap[word] &=
                                            cpu_to_le32(~mask);
                                else
                                        dp->pmap[word] |=
                                            cpu_to_le32(mask);

                                word += 1;
                        } else {
                                /* one or more words are to have all
                                 * their bits updated.  determine how
                                 * many words and how many bits.
                                 */
                                nwords = rbits >> L2DBWORD;
                                nbits = nwords << L2DBWORD;

                                /* update (free or allocate) the bits
```

```
                                 * in these words.
                                 */
                                if (free)
                                        memset(&dp->pmap[word], 0,
                                               nwords * 4);
                                else
                                        memset(&dp->pmap[word], (int) ONES,
                                               nwords * 4);

                                word += nwords;
                        }
                }

                /*
                 * update dmap lsn
                 */
                if (lblkno == lastlblkno)
                        continue;

                lastlblkno = lblkno;

                if (mp->lsn != 0) {
                        /* inherit older/smaller lsn */
                        logdiff(diffp, mp->lsn, log);
                        if (difft < diffp) {
                                mp->lsn = lsn;

                                /* move bp after tblock in logsync list */
                                LOGSYNC_LOCK(log);
                                list_del(&mp->synclist);
                                list_add(&mp->synclist, &tblk->synclist);
                                LOGSYNC_UNLOCK(log);
                        }

                        /* inherit younger/larger clsn */
                        LOGSYNC_LOCK(log);
                        logdiff(difft, tblk->clsn, log);
                        logdiff(diffp, mp->clsn, log);
                        if (difft > diffp)
                                mp->clsn = tblk->clsn;
                        LOGSYNC_UNLOCK(log);
                } else {
                        mp->log = log;
                        mp->lsn = lsn;

                        /* insert bp after tblock in logsync list */
                        LOGSYNC_LOCK(log);

                        log->count++;
                        list_add(&mp->synclist, &tblk->synclist);

                        mp->clsn = tblk->clsn;
                        LOGSYNC_UNLOCK(log);
                }
        }

        /* write the last buffer. */
        if (mp) {
                write_metapage(mp);
        }

        return (0);
}


/*
 * NAME:        dbNextAG()
 *
 * FUNCTION:    find the preferred allocation group for new allocations.
 *
 *              Within the allocation groups, we maintain a preferred
 *              allocation group which consists of a group with at least
 *              average free space.  It is the preferred group that we target
 *              new inode allocation towards.  The tie-in between inode
 *              allocation and block allocation occurs as we allocate the
 *              first (data) block of an inode and specify the inode (block)
 *              as the allocation hint for this block.
 *
 *              We try to avoid having more than one open file growing in
 *              an allocation group, as this will lead to fragmentation.
 *              This differs from the old OS/2 method of trying to keep
 *              empty ags around for large allocations.
 *
 * PARAMETERS:
 *      ipbmap  -  pointer to in-core inode for the block map.
 *
 * RETURN VALUES:
 *      the preferred allocation group number.
 */
int dbNextAG(struct inode *ipbmap)
```

```
631 {
632         s64 avgfree;
633         int agpref;
634         s64 hwm = 0;
635         int i;
636         int next_best = -1;
637         struct bmap *bmp = JFS_SBI(ipbmap->i_sb)->bmap;
638
639         BMAP_LOCK(bmp);
640
641         /* determine the average number of free blocks within the ags. */
642         avgfree = (u32)bmp->db_nfree / bmp->db_numag;
643
644         /*
645          * if the current preferred ag does not have an active allocator
646          * and has at least average freespace, return it
647          */
648         agpref = bmp->db_agpref;
649         if ((atomic_read(&bmp->db_active[agpref]) == 0) &&
650             (bmp->db_agfree[agpref] >= avgfree))
651                 goto unlock;
652
653         /* From the last preferred ag, find the next one with at least
654          * average free space.
655          */
656         for (i = 0 ; i < bmp->db_numag; i++, agpref++) {
657                 if (agpref == bmp->db_numag)
658                         agpref = 0;
659
660                 if (atomic_read(&bmp->db_active[agpref]))
661                         /* open file is currently growing in this ag */
662                         continue;
663                 if (bmp->db_agfree[agpref] >= avgfree) {
664                         /* Return this one */
665                         bmp->db_agpref = agpref;
666                         goto unlock;
667                 } else if (bmp->db_agfree[agpref] > hwm) {
668                         /* Less than avg. freespace, but best so far */
669                         hwm = bmp->db_agfree[agpref];
670                         next_best = agpref;
671                 }
672         }
673
674         /*
675          * If no inactive ag was found with average freespace, use the
676          * next best
677          */
678         if (next_best != -1)
679                 bmp->db_agpref = next_best;
680         /* else leave db_agpref unchanged */
681 unlock:
682         BMAP_UNLOCK(bmp);
683
684         /* return the preferred group.
685          */
686         return (bmp->db_agpref);
687 }
688
689 /*
690  * NAME:        dbAlloc()
691  *
692  * FUNCTION:    attempt to allocate a specified number of contiguous free
693  *              blocks from the working allocation block map.
694  *
695  *              the block allocation policy uses hints and a multi-step
696  *              approach.
697  *
698  *              for allocation requests smaller than the number of blocks
699  *              per dmap, we first try to allocate the new blocks
700  *              immediately following the hint.  if these blocks are not
701  *              available, we try to allocate blocks near the hint.  if
702  *              no blocks near the hint are available, we next try to
703  *              allocate within the same dmap as contains the hint.
704  *
705  *              if no blocks are available in the dmap or the allocation
706  *              request is larger than the dmap size, we try to allocate
707  *              within the same allocation group as contains the hint. if
708  *              this does not succeed, we finally try to allocate anywhere
709  *              within the aggregate.
710  *
711  *              we also try to allocate anywhere within the aggregate for
712  *              for allocation requests larger than the allocation group
713  *              size or requests that specify no hint value.
714  *
715  * PARAMETERS:
716  *      ip      -  pointer to in-core inode;
717  *      hint    - allocation hint.
718  *      nblocks - number of contiguous blocks in the range.
719  *      results - on successful return, set to the starting block number
720  *                of the newly allocated contiguous range.
```

```
721   *
722   * RETURN VALUES:
723   *      0       − success
724   *      ENOSPC  − insufficient disk resources
725   *      EIO     − i/o error
726   */
727  int dbAlloc(struct inode *ip, s64 hint, s64 nblocks, s64 * results)
728  {
729          int rc, agno;
730          struct inode *ipbmap = JFS_SBI(ip->i_sb)->ipbmap;
731          struct bmap *bmp;
732          struct metapage *mp;
733          s64 lblkno, blkno;
734          struct dmap *dp;
735          int l2nb;
736          s64 mapSize;
737          int writers;
738
739          /* assert that nblocks is valid */
740          assert(nblocks > 0);
741
742  #ifdef _STILL_TO_PORT
743          /* DASD limit check                                    F226941 */
744          if (OVER_LIMIT(ip, nblocks))
745                  return ENOSPC;
746  #endif                          /* _STILL_TO_PORT */
747
748          /* get the log2 number of blocks to be allocated.
749           * if the number of blocks is not a log2 multiple,
750           * it will be rounded up to the next log2 multiple.
751           */
752          l2nb = BLKSTOL2(nblocks);
753
754          bmp = JFS_SBI(ip->i_sb)->bmap;
755
756  //retry:        /* serialize w.r.t.extendfs() */
757          mapSize = bmp->db_mapsize;
758
759          /* the hint should be within the map */
760          assert(hint < mapSize);
761
762          /* if the number of blocks to be allocated is greater than the
763           * allocation group size, try to allocate anywhere.
764           */
765          if (l2nb > bmp->db_agl2size) {
766                  IWRITE_LOCK(ipbmap);
767
768                  rc = dbAllocAny(bmp, nblocks, l2nb, results);
769                  if (rc == 0) {
770                          DBALLOC(bmp->db_DBmap, bmp->db_mapsize, *results,
771                                  nblocks);
772                  }
773
774                  goto write_unlock;
775          }
776
777          /*
778           * If no hint, let dbNextAG recommend an allocation group
779           */
780          if (hint == 0)
781                  goto pref_ag;
782
783          /* we would like to allocate close to the hint.  adjust the
784           * hint to the block following the hint since the allocators
785           * will start looking for free space starting at this point.
786           */
787          blkno = hint + 1;
788
789          if (blkno >= bmp->db_mapsize)
790                  goto pref_ag;
791
792          agno = blkno >> bmp->db_agl2size;
793
794          /* check if blkno crosses over into a new allocation group.
795           * if so, check if we should allow allocations within this
796           * allocation group.
797           */
798          if ((blkno & (bmp->db_agsize − 1)) == 0)
799                  /* check if the AG is currenly being written to.
800                   * if so, call dbNextAG() to find a non−busy
801                   * AG with sufficient free space.
802                   */
803                  if (atomic_read(&bmp->db_active[agno]))
804                          goto pref_ag;
805
806          /* check if the allocation request size can be satisfied from a
807           * single dmap.  if so, try to allocate from the dmap containing
808           * the hint using a tiered strategy.
809           */
810          if (nblocks <= BPERDMAP) {
```

```
811                     IREAD_LOCK(ipbmap);
812
813                     /* get the buffer for the dmap containing the hint.
814                      */
815                     rc = EIO;
816                     lblkno = BLKTODMAP(blkno, bmp->db_l2nbperpage);
817                     mp = read_metapage(ipbmap, lblkno, PSIZE, 0);
818                     if (mp == NULL)
819                             goto read_unlock;
820
821                     dp = (struct dmap *) mp->data;
822
823                     /* first, try to satisfy the allocation request with the
824                      * blocks beginning at the hint.
825                      */
826                     if ((rc = dbAllocNext(bmp, dp, blkno, (int) nblocks))
827                         != ENOSPC) {
828                             if (rc == 0) {
829                                     *results = blkno;
830                                     DBALLOC(bmp->db_DBmap, bmp->db_mapsize,
831                                             *results, nblocks);
832                                     mark_metapage_dirty(mp);
833                             }
834
835                             release_metapage(mp);
836                             goto read_unlock;
837                     }
838
839                     writers = atomic_read(&bmp->db_active[agno]);
840                     if ((writers > 1) ||
841                         ((writers == 1) && (JFS_IP(ip)->active_ag != agno))) {
842                             /*
843                              * Someone else is writing in this allocation
844                              * group.  To avoid fragmenting, try another ag
845                              */
846                             release_metapage(mp);
847                             IREAD_UNLOCK(ipbmap);
848                             goto pref_ag;
849                     }
850
851                     /* next, try to satisfy the allocation request with blocks
852                      * near the hint.
853                      */
854                     if ((rc =
855                          dbAllocNear(bmp, dp, blkno, (int) nblocks, l2nb, results))
856                         != ENOSPC) {
857                             if (rc == 0) {
858                                     DBALLOC(bmp->db_DBmap, bmp->db_mapsize,
859                                             *results, nblocks);
860                                     mark_metapage_dirty(mp);
861                             }
862
863                             release_metapage(mp);
864                             goto read_unlock;
865                     }
866
867                     /* try to satisfy the allocation request with blocks within
868                      * the same dmap as the hint.
869                      */
870                     if ((rc = dbAllocDmapLev(bmp, dp, (int) nblocks, l2nb, results))
871                         != ENOSPC) {
872                             if (rc == 0) {
873                                     DBALLOC(bmp->db_DBmap, bmp->db_mapsize,
874                                             *results, nblocks);
875                                     mark_metapage_dirty(mp);
876                             }
877
878                             release_metapage(mp);
879                             goto read_unlock;
880                     }
881
882                     release_metapage(mp);
883                     IREAD_UNLOCK(ipbmap);
884             }
885
886             /* try to satisfy the allocation request with blocks within
887              * the same allocation group as the hint.
888              */
889             IWRITE_LOCK(ipbmap);
890             if ((rc = dbAllocAG(bmp, agno, nblocks, l2nb, results))
891                 != ENOSPC) {
892                     if (rc == 0)
893                             DBALLOC(bmp->db_DBmap, bmp->db_mapsize,
894                                     *results, nblocks);
895                     goto write_unlock;
896             }
897             IWRITE_UNLOCK(ipbmap);
898
899
900     pref_ag:
```

```
901           /*
902            * Let dbNextAG recommend a preferred allocation group
903            */
904           agno = dbNextAG(ipbmap);
905           IWRITE_LOCK(ipbmap);
906
907           /* Try to allocate within this allocation group.  if that fails, try to
908            * allocate anywhere in the map.
909            */
910           if ((rc = dbAllocAG(bmp, agno, nblocks, l2nb, results)) == ENOSPC)
911                   rc = dbAllocAny(bmp, nblocks, l2nb, results);
912           if (rc == 0) {
913                   DBALLOC(bmp->db_DBmap, bmp->db_mapsize, *results, nblocks);
914           }
915
916         write_unlock:
917           IWRITE_UNLOCK(ipbmap);
918
919           return (rc);
920
921         read_unlock:
922           IREAD_UNLOCK(ipbmap);
923
924           return (rc);
925   }
926
927
928   /*
929    * NAME:        dbAllocExact()
930    *
931    * FUNCTION:    try to allocate the requested extent;
932    *
933    * PARAMETERS:
934    *      ip      - pointer to in-core inode;
935    *      blkno   - extent address;
936    *      nblocks - extent length;
937    *
938    * RETURN VALUES:
939    *      0       - success
940    *      ENOSPC  - insufficient disk resources
941    *      EIO     - i/o error
942    */
943   int dbAllocExact(struct inode *ip, s64 blkno, int nblocks)
944   {
945           int rc;
946           struct inode *ipbmap = JFS_SBI(ip->i_sb)->ipbmap;
947           struct bmap *bmp = JFS_SBI(ip->i_sb)->bmap;
948           struct dmap *dp;
949           s64 lblkno;
950           struct metapage *mp;
951
952           IREAD_LOCK(ipbmap);
953
954           /*
955            * validate extent request:
956            *
957            * note: defragfs policy:
958            *  max 64 blocks will be moved.
959            *  allocation request size must be satisfied from a single dmap.
960            */
961           if (nblocks <= 0 || nblocks > BPERDMAP || blkno >= bmp->db_mapsize) {
962                   IREAD_UNLOCK(ipbmap);
963                   return EINVAL;
964           }
965
966           if (nblocks > ((s64) 1 << bmp->db_maxfreebud)) {
967                   /* the free space is no longer available */
968                   IREAD_UNLOCK(ipbmap);
969                   return ENOSPC;
970           }
971
972           /* read in the dmap covering the extent */
973           lblkno = BLKTODMAP(blkno, bmp->db_l2nbperpage);
974           mp = read_metapage(ipbmap, lblkno, PSIZE, 0);
975           if (mp == NULL) {
976                   IREAD_UNLOCK(ipbmap);
977                   return (EIO);
978           }
979           dp = (struct dmap *) mp->data;
980
981           /* try to allocate the requested extent */
982           rc = dbAllocNext(bmp, dp, blkno, nblocks);
983
984           IREAD_UNLOCK(ipbmap);
985
986           if (rc == 0) {
987                   DBALLOC(bmp->db_DBmap, bmp->db_mapsize, blkno, nblocks);
988                   mark_metapage_dirty(mp);
989           }
990           release_metapage(mp);
```

```
991
992             return (rc);
993     }
994
995
996     /*
997      * NAME:        dbReAlloc()
998      *
999      * FUNCTION:    attempt to extend a current allocation by a specified
1000     *              number of blocks.
1001     *
1002     *              this routine attempts to satisfy the allocation request
1003     *              by first trying to extend the existing allocation in
1004     *              place by allocating the additional blocks as the blocks
1005     *              immediately following the current allocation.  if these
1006     *              blocks are not available, this routine will attempt to
1007     *              allocate a new set of contiguous blocks large enough
1008     *              to cover the existing allocation plus the additional
1009     *              number of blocks required.
1010     *
1011     * PARAMETERS:
1012     *      ip          −  pointer to in−core inode requiring allocation.
1013     *      blkno       −  starting block of the current allocation.
1014     *      nblocks     −  number of contiguous blocks within the current
1015     *                     allocation.
1016     *      addnblocks  −  number of blocks to add to the allocation.
1017     *      results −      on successful return, set to the starting block number
1018     *                     of the existing allocation if the existing allocation
1019     *                     was extended in place or to a newly allocated contiguous
1020     *                     range if the existing allocation could not be extended
1021     *                     in place.
1022     *
1023     * RETURN VALUES:
1024     *      0       − success
1025     *      ENOSPC  − insufficient disk resources
1026     *      EIO     − i/o error
1027     */
1028    int
1029    dbReAlloc(struct inode *ip,
1030              s64 blkno, s64 nblocks, s64 addnblocks, s64 * results)
1031    {
1032            int rc;
1033
1034            /* try to extend the allocation in place.
1035             */
1036            if ((rc = dbExtend(ip, blkno, nblocks, addnblocks)) == 0) {
1037                    *results = blkno;
1038                    return (0);
1039            } else {
1040                    if (rc != ENOSPC)
1041                            return (rc);
1042            }
1043
1044            /* could not extend the allocation in place, so allocate a
1045             * new set of blocks for the entire request (i.e. try to get
1046             * a range of contiguous blocks large enough to cover the
1047             * existing allocation plus the additional blocks.)
1048             */
1049            return (dbAlloc
1050                    (ip, blkno + nblocks − 1, addnblocks + nblocks, results));
1051    }
1052
1053
1054    /*
1055     * NAME:        dbExtend()
1056     *
1057     * FUNCTION:    attempt to extend a current allocation by a specified
1058     *              number of blocks.
1059     *
1060     *              this routine attempts to satisfy the allocation request
1061     *              by first trying to extend the existing allocation in
1062     *              place by allocating the additional blocks as the blocks
1063     *              immediately following the current allocation.
1064     *
1065     * PARAMETERS:
1066     *      ip          −  pointer to in−core inode requiring allocation.
1067     *      blkno       −  starting block of the current allocation.
1068     *      nblocks     −  number of contiguous blocks within the current
1069     *                     allocation.
1070     *      addnblocks  −  number of blocks to add to the allocation.
1071     *
1072     * RETURN VALUES:
1073     *      0       − success
1074     *      ENOSPC  − insufficient disk resources
1075     *      EIO     − i/o error
1076     */
1077    int dbExtend(struct inode *ip, s64 blkno, s64 nblocks, s64 addnblocks)
1078    {
1079            struct jfs_sb_info *sbi = JFS_SBI(ip−>i_sb);
1080            s64 lblkno, lastblkno, extblkno;
```

```
1081          uint rel_block;
1082          struct metapage *mp;
1083          struct dmap *dp;
1084          int rc;
1085          struct inode *ipbmap = sbi->ipbmap;
1086          struct bmap *bmp;
1087
1088          /*
1089           * We don't want a non-aligned extent to cross a page boundary
1090           */
1091          if (((rel_block = blkno & (sbi->nbperpage - 1))) &&
1092              (rel_block + nblocks + addnblocks > sbi->nbperpage))
1093                  return (ENOSPC);
1094
1095          /* get the last block of the current allocation */
1096          lastblkno = blkno + nblocks - 1;
1097
1098          /* determine the block number of the block following
1099           * the existing allocation.
1100           */
1101          extblkno = lastblkno + 1;
1102
1103          IREAD_LOCK(ipbmap);
1104
1105          /* better be within the file system */
1106          bmp = sbi->bmap;
1107          assert(lastblkno >= 0 && lastblkno < bmp->db_mapsize);
1108
1109          /* we'll attempt to extend the current allocation in place by
1110           * allocating the additional blocks as the blocks immediately
1111           * following the current allocation.  we only try to extend the
1112           * current allocation in place if the number of additional blocks
1113           * can fit into a dmap, the last block of the current allocation
1114           * is not the last block of the file system, and the start of the
1115           * inplace extension is not on an allocation group boundry.
1116           */
1117          if (addnblocks > BPERDMAP || extblkno >= bmp->db_mapsize ||
1118              (extblkno & (bmp->db_agsize - 1)) == 0) {
1119                  IREAD_UNLOCK(ipbmap);
1120                  return (ENOSPC);
1121          }
1122
1123          /* get the buffer for the dmap containing the first block
1124           * of the extension.
1125           */
1126          lblkno = BLKTODMAP(extblkno, bmp->db_l2nbperpage);
1127          mp = read_metapage(ipbmap, lblkno, PSIZE, 0);
1128          if (mp == NULL) {
1129                  IREAD_UNLOCK(ipbmap);
1130                  return (EIO);
1131          }
1132
1133          DBALLOCCK(bmp->db_DBmap, bmp->db_mapsize, blkno, nblocks);
1134          dp = (struct dmap *) mp->data;
1135
1136          /* try to allocate the blocks immediately following the
1137           * current allocation.
1138           */
1139          rc = dbAllocNext(bmp, dp, extblkno, (int) addnblocks);
1140
1141          IREAD_UNLOCK(ipbmap);
1142
1143          /* were we successful ? */
1144          if (rc == 0) {
1145                  DBALLOC(bmp->db_DBmap, bmp->db_mapsize, extblkno,
1146                          addnblocks);
1147                  write_metapage(mp);
1148          } else {
1149                  /* we were not successful */
1150                  release_metapage(mp);
1151                  assert(rc == ENOSPC || rc == EIO);
1152          }
1153
1154          return (rc);
1155  }
1156
1157
1158  /*
1159   * NAME:        dbAllocNext()
1160   *
1161   * FUNCTION:    attempt to allocate the blocks of the specified block
1162   *              range within a dmap.
1163   *
1164   * PARAMETERS:
1165   *      bmp     -  pointer to bmap descriptor
1166   *      dp      -  pointer to dmap.
1167   *      blkno   -  starting block number of the range.
1168   *      nblocks -  number of contiguous free blocks of the range.
1169   *
1170   * RETURN VALUES:
```

```
 *      0       − success
 *      ENOSPC  − insufficient disk resources
 *      EIO     − i/o error
 *
 * serialization: IREAD_LOCK(ipbmap) held on entry/exit;
 */
static int dbAllocNext(struct bmap * bmp, struct dmap * dp, s64 blkno,
                       int nblocks)
{
        int dbitno, word, rembits, nb, nwords, wbitno, nw;
        int l2size;
        s8 *leaf;
        u32 mask;

        /* pick up a pointer to the leaves of the dmap tree.
         */
        leaf = dp->tree.stree + le32_to_cpu(dp->tree.leafidx);

        /* determine the bit number and word within the dmap of the
         * starting block.
         */
        dbitno = blkno & (BPERDMAP − 1);
        word = dbitno >> L2DBWORD;

        /* check if the specified block range is contained within
         * this dmap.
         */
        if (dbitno + nblocks > BPERDMAP)
                return (ENOSPC);

        /* check if the starting leaf indicates that anything
         * is free.
         */
        if (leaf[word] == NOFREE)
                return (ENOSPC);

        /* check the dmaps words corresponding to block range to see
         * if the block range is free.  not all bits of the first and
         * last words may be contained within the block range.  if this
         * is the case, we'll work against those words (i.e. partial first
         * and/or last) on an individual basis (a single pass) and examine
         * the actual bits to determine if they are free.  a single pass
         * will be used for all dmap words fully contained within the
         * specified range.  within this pass, the leaves of the dmap
         * tree will be examined to determine if the blocks are free. a
         * single leaf may describe the free space of multiple dmap
         * words, so we may visit only a subset of the actual leaves
         * corresponding to the dmap words of the block range.
         */
        for (rembits = nblocks; rembits > 0; rembits −= nb, dbitno += nb) {
                /* determine the bit number within the word and
                 * the number of bits within the word.
                 */
                wbitno = dbitno & (DBWORD − 1);
                nb = min(rembits, DBWORD − wbitno);

                /* check if only part of the word is to be examined.
                 */
                if (nb < DBWORD) {
                        /* check if the bits are free.
                         */
                        mask = (ONES << (DBWORD − nb) >> wbitno);
                        if ((mask & ~le32_to_cpu(dp->wmap[word])) != mask)
                                return (ENOSPC);

                        word += 1;
                } else {
                        /* one or more dmap words are fully contained
                         * within the block range.  determine how many
                         * words and how many bits.
                         */
                        nwords = rembits >> L2DBWORD;
                        nb = nwords << L2DBWORD;

                        /* now examine the appropriate leaves to determine
                         * if the blocks are free.
                         */
                        while (nwords > 0) {
                                /* does the leaf describe any free space ?
                                 */
                                if (leaf[word] < BUDMIN)
                                        return (ENOSPC);

                                /* determine the l2 number of bits provided
                                 * by this leaf.
                                 */
                                l2size =
                                    min((int)leaf[word], NLSTOL2BSZ(nwords));

                                /* determine how many words were handled.
```

```
1261                                    */
1262                                   nw = BUDSIZE(l2size, BUDMIN);
1263
1264                                   nwords −= nw;
1265                                   word += nw;
1266                           }
1267                   }
1268           }
1269
1270           /* allocate the blocks.
1271            */
1272           return (dbAllocDmap(bmp, dp, blkno, nblocks));
1273   }
1274
1275
1276   /*
1277    * NAME:        dbAllocNear()
1278    *
1279    * FUNCTION:    attempt to allocate a number of contiguous free blocks near
1280    *              a specified block (hint) within a dmap.
1281    *
1282    *              starting with the dmap leaf that covers the hint, we'll
1283    *              check the next four contiguous leaves for sufficient free
1284    *              space.  if sufficient free space is found, we'll allocate
1285    *              the desired free space.
1286    *
1287    * PARAMETERS:
1288    *      bmp     −  pointer to bmap descriptor
1289    *      dp      −  pointer to dmap.
1290    *      blkno   −  block number to allocate near.
1291    *      nblocks −  actual number of contiguous free blocks desired.
1292    *      l2nb    −  log2 number of contiguous free blocks desired.
1293    *      results −  on successful return, set to the starting block number
1294    *                 of the newly allocated range.
1295    *
1296    * RETURN VALUES:
1297    *      0       − success
1298    *      ENOSPC  − insufficient disk resources
1299    *      EIO     − i/o error
1300    *
1301    * serialization: IREAD_LOCK(ipbmap) held on entry/exit;
1302    */
1303   static int
1304   dbAllocNear(struct bmap * bmp,
1305               struct dmap * dp, s64 blkno, int nblocks, int l2nb, s64 * results)
1306   {
1307           int word, lword, rc;
1308           s8 *leaf = dp−>tree.stree + le32_to_cpu(dp−>tree.leafidx);
1309
1310           /* determine the word within the dmap that holds the hint
1311            * (i.e. blkno).  also, determine the last word in the dmap
1312            * that we'll include in our examination.
1313            */
1314           word = (blkno & (BPERDMAP − 1)) >> L2DBWORD;
1315           lword = min(word + 4, LPERDMAP);
1316
1317           /* examine the leaves for sufficient free space.
1318            */
1319           for (; word < lword; word++) {
1320                   /* does the leaf describe sufficient free space ?
1321                    */
1322                   if (leaf[word] < l2nb)
1323                           continue;
1324
1325                   /* determine the block number within the file system
1326                    * of the first block described by this dmap word.
1327                    */
1328                   blkno = le64_to_cpu(dp−>start) + (word << L2DBWORD);
1329
1330                   /* if not all bits of the dmap word are free, get the
1331                    * starting bit number within the dmap word of the required
1332                    * string of free bits and adjust the block number with the
1333                    * value.
1334                    */
1335                   if (leaf[word] < BUDMIN)
1336                           blkno +=
1337                               dbFindBits(le32_to_cpu(dp−>wmap[word]), l2nb);
1338
1339                   /* allocate the blocks.
1340                    */
1341                   if ((rc = dbAllocDmap(bmp, dp, blkno, nblocks)) == 0)
1342                           *results = blkno;
1343
1344                   return (rc);
1345           }
1346
1347           return (ENOSPC);
1348   }
1349
1350
```

```
1351 /*
1352  * NAME:        dbAllocAG()
1353  *
1354  * FUNCTION:    attempt to allocate the specified number of contiguous
1355  *              free blocks within the specified allocation group.
1356  *
1357  *              unless the allocation group size is equal to the number
1358  *              of blocks per dmap, the dmap control pages will be used to
1359  *              find the required free space, if available.  we start the
1360  *              search at the highest dmap control page level which
1361  *              distinctly describes the allocation group's free space
1362  *              (i.e. the highest level at which the allocation group's
1363  *              free space is not mixed in with that of any other group).
1364  *              in addition, we start the search within this level at a
1365  *              height of the dmapctl dmtree at which the nodes distinctly
1366  *              describe the allocation group's free space.  at this height,
1367  *              the allocation group's free space may be represented by 1
1368  *              or two sub−trees, depending on the allocation group size.
1369  *              we search the top nodes of these subtrees left to right for
1370  *              sufficient free space.  if sufficient free space is found,
1371  *              the subtree is searched to find the leftmost leaf that
1372  *              has free space.  once we have made it to the leaf, we
1373  *              move the search to the next lower level dmap control page
1374  *              corresponding to this leaf.  we continue down the dmap control
1375  *              pages until we find the dmap that contains or starts the
1376  *              sufficient free space and we allocate at this dmap.
1377  *
1378  *              if the allocation group size is equal to the dmap size,
1379  *              we'll start at the dmap corresponding to the allocation
1380  *              group and attempt the allocation at this level.
1381  *
1382  *              the dmap control page search is also not performed if the
1383  *              allocation group is completely free and we go to the first
1384  *              dmap of the allocation group to do the allocation.  this is
1385  *              done because the allocation group may be part (not the first
1386  *              part) of a larger binary buddy system, causing the dmap
1387  *              control pages to indicate no free space (NOFREE) within
1388  *              the allocation group.
1389  *
1390  * PARAMETERS:
1391  *      bmp     −  pointer to bmap descriptor
1392  *      agno    − allocation group number.
1393  *      nblocks −  actual number of contiguous free blocks desired.
1394  *      l2nb    −  log2 number of contiguous free blocks desired.
1395  *      results −  on successful return, set to the starting block number
1396  *                 of the newly allocated range.
1397  *
1398  * RETURN VALUES:
1399  *      0       − success
1400  *      ENOSPC  − insufficient disk resources
1401  *      EIO     − i/o error
1402  *
1403  * note: IWRITE_LOCK(ipmap) held on entry/exit;
1404  */
1405 static int
1406 dbAllocAG(struct bmap * bmp, int agno, s64 nblocks, int l2nb, s64 * results)
1407 {
1408         struct metapage *mp;
1409         struct dmapctl *dcp;
1410         int rc, ti, i, k, m, n, agperlev;
1411         s64 blkno, lblkno;
1412         int budmin;
1413
1414         /* allocation request should not be for more than the
1415          * allocation group size.
1416          */
1417         assert(l2nb <= bmp->db_agl2size);
1418
1419         /* determine the starting block number of the allocation
1420          * group.
1421          */
1422         blkno = (s64) agno << bmp->db_agl2size;
1423
1424         /* check if the allocation group size is the minimum allocation
1425          * group size or if the allocation group is completely free. if
1426          * the allocation group size is the minimum size of BPERDMAP (i.e.
1427          * 1 dmap), there is no need to search the dmap control page (below)
1428          * that fully describes the allocation group since the allocation
1429          * group is already fully described by a dmap.  in this case, we
1430          * just call dbAllocCtl() to search the dmap tree and allocate the
1431          * required space if available.
1432          *
1433          * if the allocation group is completely free, dbAllocCtl() is
1434          * also called to allocate the required space.  this is done for
1435          * two reasons.  first, it makes no sense searching the dmap control
1436          * pages for free space when we know that free space exists.  second,
1437          * the dmap control pages may indicate that the allocation group
1438          * has no free space if the allocation group is part (not the first
1439          * part) of a larger binary buddy system.
1440          */
```

```
1441            if (bmp->db_agsize == BPERDMAP
1442                || bmp->db_agfree[agno] == bmp->db_agsize) {
1443                    rc = dbAllocCtl(bmp, nblocks, l2nb, blkno, results);
1444                    /* assert(!(rc == ENOSPC && bmp->db_agfree[agno] == bmp->db_agsize)); */
1445                    if ((rc == ENOSPC) &&
1446                        (bmp->db_agfree[agno] == bmp->db_agsize)) {
1447                            jERROR(1,
1448                                   ("dbAllocAG: removed assert, but still need to debug here\nblkno = 0x%Lx, nblocks = 0x%Lx\n",
1449                                    (unsigned long long) blkno,
1450                                    (unsigned long long) nblocks));
1451                    }
1452                    return (rc);
1453            }
1454
1455            /* the buffer for the dmap control page that fully describes the
1456             * allocation group.
1457             */
1458            lblkno = BLKTOCTL(blkno, bmp->db_l2nbperpage, bmp->db_aglevel);
1459            mp = read_metapage(bmp->db_ipbmap, lblkno, PSIZE, 0);
1460            if (mp == NULL)
1461                    return (EIO);
1462            dcp = (struct dmapctl *) mp->data;
1463            budmin = dcp->budmin;
1464
1465            /* search the subtree(s) of the dmap control page that describes
1466             * the allocation group, looking for sufficient free space.  to begin,
1467             * determine how many allocation groups are represented in a dmap
1468             * control page at the control page level (i.e. L0, L1, L2) that
1469             * fully describes an allocation group. next, determine the starting
1470             * tree index of this allocation group within the control page.
1471             */
1472            agperlev =
1473                (1 << (L2LPERCTL - (bmp->db_agheigth << 1))) / bmp->db_agwidth;
1474            ti = bmp->db_agstart + bmp->db_agwidth * (agno & (agperlev - 1));
1475
1476            /* dmap control page trees fan-out by 4 and a single allocation
1477             * group may be described by 1 or 2 subtrees within the ag level
1478             * dmap control page, depending upon the ag size. examine the ag's
1479             * subtrees for sufficient free space, starting with the leftmost
1480             * subtree.
1481             */
1482            for (i = 0; i < bmp->db_agwidth; i++, ti++) {
1483                    /* is there sufficient free space ?
1484                     */
1485                    if (l2nb > dcp->stree[ti])
1486                            continue;
1487
1488                    /* sufficient free space found in a subtree. now search down
1489                     * the subtree to find the leftmost leaf that describes this
1490                     * free space.
1491                     */
1492                    for (k = bmp->db_agheigth; k > 0; k--) {
1493                            for (n = 0, m = (ti << 2) + 1; n < 4; n++) {
1494                                    if (l2nb <= dcp->stree[m + n]) {
1495                                            ti = m + n;
1496                                            break;
1497                                    }
1498                            }
1499                            assert(n < 4);
1500                    }
1501
1502                    /* determine the block number within the file system
1503                     * that corresponds to this leaf.
1504                     */
1505                    if (bmp->db_aglevel == 2)
1506                            blkno = 0;
1507                    else if (bmp->db_aglevel == 1)
1508                            blkno &= ~(MAXL1SIZE - 1);
1509                    else            /* bmp->db_aglevel == 0 */
1510                            blkno &= ~(MAXL0SIZE - 1);
1511
1512                    blkno +=
1513                        ((s64) (ti - le32_to_cpu(dcp->leafidx))) << budmin;
1514
1515                    /* release the buffer in preparation for going down
1516                     * the next level of dmap control pages.
1517                     */
1518                    release_metapage(mp);
1519
1520                    /* check if we need to continue to search down the lower
1521                     * level dmap control pages.  we need to if the number of
1522                     * blocks required is less than maximum number of blocks
1523                     * described at the next lower level.
1524                     */
1525                    if (l2nb < budmin) {
1526
1527                            /* search the lower level dmap control pages to get
1528                             * the starting block number of the the dmap that
1529                             * contains or starts off the free space.
1530                             */
```

```
1531                            if ((rc =
1532                                 dbFindCtl(bmp, l2nb, bmp->db_aglevel - 1,
1533                                           &blkno))) {
1534                                    assert(rc != ENOSPC);
1535                                    return (rc);
1536                            }
1537                    }
1538
1539                    /* allocate the blocks.
1540                     */
1541                    rc = dbAllocCtl(bmp, nblocks, l2nb, blkno, results);
1542                    assert(rc != ENOSPC);
1543                    return (rc);
1544            }
1545
1546            /* no space in the allocation group.  release the buffer and
1547             * return ENOSPC.
1548             */
1549            release_metapage(mp);
1550
1551            return (ENOSPC);
1552    }
1553
1554
1555    /*
1556     * NAME:        dbAllocAny()
1557     *
1558     * FUNCTION:    attempt to allocate the specified number of contiguous
1559     *              free blocks anywhere in the file system.
1560     *
1561     *              dbAllocAny() attempts to find the sufficient free space by
1562     *              searching down the dmap control pages, starting with the
1563     *              highest level (i.e. L0, L1, L2) control page.  if free space
1564     *              large enough to satisfy the desired free space is found, the
1565     *              desired free space is allocated.
1566     *
1567     * PARAMETERS:
1568     *      bmp     -  pointer to bmap descriptor
1569     *      nblocks  -  actual number of contiguous free blocks desired.
1570     *      l2nb     -  log2 number of contiguous free blocks desired.
1571     *      results -  on successful return, set to the starting block number
1572     *                 of the newly allocated range.
1573     *
1574     * RETURN VALUES:
1575     *      0       - success
1576     *      ENOSPC  - insufficient disk resources
1577     *      EIO     - i/o error
1578     *
1579     * serialization: IWRITE_LOCK(ipbmap) held on entry/exit;
1580     */
1581    static int dbAllocAny(struct bmap * bmp, s64 nblocks, int l2nb, s64 * results)
1582    {
1583            int rc;
1584            s64 blkno = 0;
1585
1586            /* starting with the top level dmap control page, search
1587             * down the dmap control levels for sufficient free space.
1588             * if free space is found, dbFindCtl() returns the starting
1589             * block number of the dmap that contains or starts off the
1590             * range of free space.
1591             */
1592            if ((rc = dbFindCtl(bmp, l2nb, bmp->db_maxlevel, &blkno)))
1593                    return (rc);
1594
1595            /* allocate the blocks.
1596             */
1597            rc = dbAllocCtl(bmp, nblocks, l2nb, blkno, results);
1598            assert(rc != ENOSPC);
1599            return (rc);
1600    }
1601
1602
1603    /*
1604     * NAME:        dbFindCtl()
1605     *
1606     * FUNCTION:    starting at a specified dmap control page level and block
1607     *              number, search down the dmap control levels for a range of
1608     *              contiguous free blocks large enough to satisfy an allocation
1609     *              request for the specified number of free blocks.
1610     *
1611     *              if sufficient contiguous free blocks are found, this routine
1612     *              returns the starting block number within a dmap page that
1613     *              contains or starts a range of contiqious free blocks that
1614     *              is sufficient in size.
1615     *
1616     * PARAMETERS:
1617     *      bmp     -  pointer to bmap descriptor
1618     *      level   -  starting dmap control page level.
1619     *      l2nb    -  log2 number of contiguous free blocks desired.
1620     *      *blkno  -  on entry, starting block number for conducting the search.
```

```
1621  *                 on successful return, the first block within a dmap page
1622  *                 that contains or starts a range of contiguous free blocks.
1623  *
1624  * RETURN VALUES:
1625  *      0       − success
1626  *      ENOSPC  − insufficient disk resources
1627  *      EIO     − i/o error
1628  *
1629  * serialization: IWRITE_LOCK(ipbmap) held on entry/exit;
1630  */
1631 static int dbFindCtl(struct bmap * bmp, int l2nb, int level, s64 * blkno)
1632 {
1633         int rc, leafidx, lev;
1634         s64 b, lblkno;
1635         struct dmapctl *dcp;
1636         int budmin;
1637         struct metapage *mp;
1638
1639         /* starting at the specified dmap control page level and block
1640          * number, search down the dmap control levels for the starting
1641          * block number of a dmap page that contains or starts off
1642          * sufficient free blocks.
1643          */
1644         for (lev = level, b = *blkno; lev >= 0; lev−−) {
1645                 /* get the buffer of the dmap control page for the block
1646                  * number and level (i.e. L0, L1, L2).
1647                  */
1648                 lblkno = BLKTOCTL(b, bmp−>db_l2nbperpage, lev);
1649                 mp = read_metapage(bmp−>db_ipbmap, lblkno, PSIZE, 0);
1650                 if (mp == NULL)
1651                         return (EIO);
1652                 dcp = (struct dmapctl *) mp−>data;
1653                 budmin = dcp−>budmin;
1654
1655                 /* search the tree within the dmap control page for
1656                  * sufficent free space.  if sufficient free space is found,
1657                  * dbFindLeaf() returns the index of the leaf at which
1658                  * free space was found.
1659                  */
1660                 rc = dbFindLeaf((dmtree_t *) dcp, l2nb, &leafidx);
1661
1662                 /* release the buffer.
1663                  */
1664                 release_metapage(mp);
1665
1666                 /* space found ?
1667                  */
1668                 if (rc) {
1669                         assert(lev == level);
1670                         return (ENOSPC);
1671                 }
1672
1673                 /* adjust the block number to reflect the location within
1674                  * the dmap control page (i.e. the leaf) at which free
1675                  * space was found.
1676                  */
1677                 b += (((s64) leafidx) << budmin);
1678
1679                 /* we stop the search at this dmap control page level if
1680                  * the number of blocks required is greater than or equal
1681                  * to the maximum number of blocks described at the next
1682                  * (lower) level.
1683                  */
1684                 if (l2nb >= budmin)
1685                         break;
1686         }
1687
1688         *blkno = b;
1689         return (0);
1690 }
1691
1692
1693 /*
1694  * NAME:        dbAllocCtl()
1695  *
1696  * FUNCTION:    attempt to allocate a specified number of contiguous
1697  *              blocks starting within a specific dmap.
1698  *
1699  *              this routine is called by higher level routines that search
1700  *              the dmap control pages above the actual dmaps for contiguous
1701  *              free space.  the result of successful searches by these
1702  *              routines are the starting block numbers within dmaps, with
1703  *              the dmaps themselves containing the desired contiguous free
1704  *              space or starting a contiguous free space of desired size
1705  *              that is made up of the blocks of one or more dmaps. these
1706  *              calls should not fail due to insufficent resources.
1707  *
1708  *              this routine is called in some cases where it is not known
1709  *              whether it will fail due to insufficient resources.  more
1710  *              specifically, this occurs when allocating from an allocation
```

```
1711  *              group whose size is equal to the number of blocks per dmap.
1712  *              in this case, the dmap control pages are not examined prior
1713  *              to calling this routine (to save pathlength) and the call
1714  *              might fail.
1715  *
1716  *              for a request size that fits within a dmap, this routine relies
1717  *              upon the dmap's dmtree to find the requested contiguous free
1718  *              space.  for request sizes that are larger than a dmap, the
1719  *              requested free space will start at the first block of the
1720  *              first dmap (i.e. blkno).
1721  *
1722  * PARAMETERS:
1723  *      bmp     −  pointer to bmap descriptor
1724  *      nblocks  −  actual number of contiguous free blocks to allocate.
1725  *      l2nb     −  log2 number of contiguous free blocks to allocate.
1726  *      blkno    −  starting block number of the dmap to start the allocation
1727  *                  from.
1728  *      results −  on successful return, set to the starting block number
1729  *                 of the newly allocated range.
1730  *
1731  * RETURN VALUES:
1732  *      0       − success
1733  *      ENOSPC  − insufficient disk resources
1734  *      EIO     − i/o error
1735  *
1736  * serialization: IWRITE_LOCK(ipbmap) held on entry/exit;
1737  */
1738 static int
1739 dbAllocCtl(struct bmap * bmp, s64 nblocks, int l2nb, s64 blkno, s64 * results)
1740 {
1741         int rc, nb;
1742         s64 b, lblkno, n;
1743         struct metapage *mp;
1744         struct dmap *dp;
1745
1746         /* check if the allocation request is confined to a single dmap.
1747          */
1748         if (l2nb <= L2BPERDMAP) {
1749                 /* get the buffer for the dmap.
1750                  */
1751                 lblkno = BLKTODMAP(blkno, bmp->db_l2nbperpage);
1752                 mp = read_metapage(bmp->db_ipbmap, lblkno, PSIZE, 0);
1753                 if (mp == NULL)
1754                         return (EIO);
1755                 dp = (struct dmap *) mp->data;
1756
1757                 /* try to allocate the blocks.
1758                  */
1759                 rc = dbAllocDmapLev(bmp, dp, (int) nblocks, l2nb, results);
1760                 if (rc == 0)
1761                         mark_metapage_dirty(mp);
1762
1763                 release_metapage(mp);
1764
1765                 return (rc);
1766         }
1767
1768         /* allocation request involving multiple dmaps. it must start on
1769          * a dmap boundary.
1770          */
1771         assert((blkno & (BPERDMAP − 1)) == 0);
1772
1773         /* allocate the blocks dmap by dmap.
1774          */
1775         for (n = nblocks, b = blkno; n > 0; n −= nb, b += nb) {
1776                 /* get the buffer for the dmap.
1777                  */
1778                 lblkno = BLKTODMAP(b, bmp->db_l2nbperpage);
1779                 mp = read_metapage(bmp->db_ipbmap, lblkno, PSIZE, 0);
1780                 if (mp == NULL) {
1781                         rc = EIO;
1782                         goto backout;
1783                 }
1784                 dp = (struct dmap *) mp->data;
1785
1786                 /* the dmap better be all free.
1787                  */
1788                 assert(dp->tree.stree[ROOT] == L2BPERDMAP);
1789
1790                 /* determine how many blocks to allocate from this dmap.
1791                  */
1792                 nb = min(n, (s64)BPERDMAP);
1793
1794                 /* allocate the blocks from the dmap.
1795                  */
1796                 if ((rc = dbAllocDmap(bmp, dp, b, nb))) {
1797                         release_metapage(mp);
1798                         goto backout;
1799                 }
1800
```

```
1801                    /* write the buffer.
1802                     */
1803                    write_metapage(mp);
1804            }
1805
1806            /* set the results (starting block number) and return.
1807             */
1808            *results = blkno;
1809            return (0);
1810
1811            /* something failed in handling an allocation request involving
1812             * multiple dmaps.  we'll try to clean up by backing out any
1813             * allocation that has already happened for this request.  if
1814             * we fail in backing out the allocation, we'll mark the file
1815             * system to indicate that blocks have been leaked.
1816             */
1817          backout:
1818
1819            /* try to backout the allocations dmap by dmap.
1820             */
1821            for (n = nblocks - n, b = blkno; n > 0;
1822                 n -= BPERDMAP, b += BPERDMAP) {
1823                    /* get the buffer for this dmap.
1824                     */
1825                    lblkno = BLKTODMAP(b, bmp->db_l2nbperpage);
1826                    mp = read_metapage(bmp->db_ipbmap, lblkno, PSIZE, 0);
1827                    if (mp == NULL) {
1828                            /* could not back out.  mark the file system
1829                             * to indicate that we have leaked blocks.
1830                             */
1831                            fsDirty();      /* !!! */
1832                            jERROR(1,
1833                                   ("dbAllocCtl: I/O Error: Block Leakage.\n"));
1834                            continue;
1835                    }
1836                    dp = (struct dmap *) mp->data;
1837
1838                    /* free the blocks is this dmap.
1839                     */
1840                    if (dbFreeDmap(bmp, dp, b, BPERDMAP)) {
1841                            /* could not back out.  mark the file system
1842                             * to indicate that we have leaked blocks.
1843                             */
1844                            release_metapage(mp);
1845                            fsDirty();      /* !!! */
1846                            jERROR(1, ("dbAllocCtl: Block Leakage.\n"));
1847                            continue;
1848                    }
1849
1850                    /* write the buffer.
1851                     */
1852                    write_metapage(mp);
1853            }
1854
1855            return (rc);
1856    }
1857
1858
1859    /*
1860     * NAME:        dbAllocDmapLev()
1861     *
1862     * FUNCTION:    attempt to allocate a specified number of contiguous blocks
1863     *              from a specified dmap.
1864     *
1865     *              this routine checks if the contiguous blocks are available.
1866     *              if so, nblocks of blocks are allocated; otherwise, ENOSPC is
1867     *              returned.
1868     *
1869     * PARAMETERS:
1870     *      mp      -  pointer to bmap descriptor
1871     *      dp      -  pointer to dmap to attempt to allocate blocks from.
1872     *      l2nb    -  log2 number of contiguous block desired.
1873     *      nblocks -  actual number of contiguous block desired.
1874     *      results -  on successful return, set to the starting block number
1875     *                 of the newly allocated range.
1876     *
1877     * RETURN VALUES:
1878     *      0       - success
1879     *      ENOSPC  - insufficient disk resources
1880     *      EIO     - i/o error
1881     *
1882     * serialization: IREAD_LOCK(ipbmap), e.g., from dbAlloc(), or
1883     *      IWRITE_LOCK(ipbmap), e.g., dbAllocCtl(), held on entry/exit;
1884     */
1885    static int
1886    dbAllocDmapLev(struct bmap * bmp,
1887                   struct dmap * dp, int nblocks, int l2nb, s64 * results)
1888    {
1889            s64 blkno;
1890            int leafidx, rc;
```

```
1891
1892            /* can't be more than a dmaps worth of blocks */
1893            assert(l2nb <= L2BPERDMAP);
1894
1895            /* search the tree within the dmap page for sufficient
1896             * free space.  if sufficient free space is found, dbFindLeaf()
1897             * returns the index of the leaf at which free space was found.
1898             */
1899            if (dbFindLeaf((dmtree_t *) & dp->tree, l2nb, &leafidx))
1900                    return (ENOSPC);
1901
1902            /* determine the block number within the file system corresponding
1903             * to the leaf at which free space was found.
1904             */
1905            blkno = le64_to_cpu(dp->start) + (leafidx << L2DBWORD);
1906
1907            /* if not all bits of the dmap word are free, get the starting
1908             * bit number within the dmap word of the required string of free
1909             * bits and adjust the block number with this value.
1910             */
1911            if (dp->tree.stree[leafidx + LEAFIND] < BUDMIN)
1912                    blkno += dbFindBits(le32_to_cpu(dp->wmap[leafidx]), l2nb);
1913
1914            /* allocate the blocks */
1915            if ((rc = dbAllocDmap(bmp, dp, blkno, nblocks)) == 0)
1916                    *results = blkno;
1917
1918            return (rc);
1919    }
1920
1921
1922    /*
1923     * NAME:        dbAllocDmap()
1924     *
1925     * FUNCTION:    adjust the disk allocation map to reflect the allocation
1926     *              of a specified block range within a dmap.
1927     *
1928     *              this routine allocates the specified blocks from the dmap
1929     *              through a call to dbAllocBits(). if the allocation of the
1930     *              block range causes the maximum string of free blocks within
1931     *              the dmap to change (i.e. the value of the root of the dmap's
1932     *              dmtree), this routine will cause this change to be reflected
1933     *              up through the appropriate levels of the dmap control pages
1934     *              by a call to dbAdjCtl() for the L0 dmap control page that
1935     *              covers this dmap.
1936     *
1937     * PARAMETERS:
1938     *      bmp     -  pointer to bmap descriptor
1939     *      dp      -  pointer to dmap to allocate the block range from.
1940     *      blkno   -  starting block number of the block to be allocated.
1941     *      nblocks -  number of blocks to be allocated.
1942     *
1943     * RETURN VALUES:
1944     *      0       - success
1945     *      EIO     - i/o error
1946     *
1947     * serialization: IREAD_LOCK(ipbmap) or IWRITE_LOCK(ipbmap) held on entry/exit;
1948     */
1949    static int dbAllocDmap(struct bmap * bmp, struct dmap * dp, s64 blkno,
1950                           int nblocks)
1951    {
1952            s8 oldroot;
1953            int rc;
1954
1955            /* save the current value of the root (i.e. maximum free string)
1956             * of the dmap tree.
1957             */
1958            oldroot = dp->tree.stree[ROOT];
1959
1960            /* allocate the specified (blocks) bits */
1961            dbAllocBits(bmp, dp, blkno, nblocks);
1962
1963            /* if the root has not changed, done. */
1964            if (dp->tree.stree[ROOT] == oldroot)
1965                    return (0);
1966
1967            /* root changed. bubble the change up to the dmap control pages.
1968             * if the adjustment of the upper level control pages fails,
1969             * backout the bit allocation (thus making everything consistent).
1970             */
1971            if ((rc = dbAdjCtl(bmp, blkno, dp->tree.stree[ROOT], 1, 0)))
1972                    dbFreeBits(bmp, dp, blkno, nblocks);
1973
1974            return (rc);
1975    }
1976
1977
1978    /*
1979     * NAME:        dbFreeDmap()
1980     *
```

```
1981  * FUNCTION:    adjust the disk allocation map to reflect the allocation
1982  *              of a specified block range within a dmap.
1983  *
1984  *              this routine frees the specified blocks from the dmap through
1985  *              a call to dbFreeBits(). if the deallocation of the block range
1986  *              causes the maximum string of free blocks within the dmap to
1987  *              change (i.e. the value of the root of the dmap's dmtree), this
1988  *              routine will cause this change to be reflected up through the
1989  *              appropriate levels of the dmap control pages by a call to
1990  *              dbAdjCtl() for the L0 dmap control page that covers this dmap.
1991  *
1992  * PARAMETERS:
1993  *      bmp     −  pointer to bmap descriptor
1994  *      dp      −  pointer to dmap to free the block range from.
1995  *      blkno   −  starting block number of the block to be freed.
1996  *      nblocks −  number of blocks to be freed.
1997  *
1998  * RETURN VALUES:
1999  *      0       − success
2000  *      EIO     − i/o error
2001  *
2002  * serialization: IREAD_LOCK(ipbmap) or IWRITE_LOCK(ipbmap) held on entry/exit;
2003  */
2004 static int dbFreeDmap(struct bmap * bmp, struct dmap * dp, s64 blkno,
2005                       int nblocks)
2006 {
2007         s8 oldroot;
2008         int rc, word;
2009
2010         /* save the current value of the root (i.e. maximum free string)
2011          * of the dmap tree.
2012          */
2013         oldroot = dp->tree.stree[ROOT];
2014
2015         /* free the specified (blocks) bits */
2016         dbFreeBits(bmp, dp, blkno, nblocks);
2017
2018         /* if the root has not changed, done. */
2019         if (dp->tree.stree[ROOT] == oldroot)
2020                 return (0);
2021
2022         /* root changed. bubble the change up to the dmap control pages.
2023          * if the adjustment of the upper level control pages fails,
2024          * backout the deallocation.
2025          */
2026         if ((rc = dbAdjCtl(bmp, blkno, dp->tree.stree[ROOT], 0, 0))) {
2027                 word = (blkno & (BPERDMAP − 1)) >> L2DBWORD;
2028
2029                 /* as part of backing out the deallocation, we will have
2030                  * to back split the dmap tree if the deallocation caused
2031                  * the freed blocks to become part of a larger binary buddy
2032                  * system.
2033                  */
2034                 if (dp->tree.stree[word] == NOFREE)
2035                         dbBackSplit((dmtree_t *) & dp->tree, word);
2036
2037                 dbAllocBits(bmp, dp, blkno, nblocks);
2038         }
2039
2040         return (rc);
2041 }
2042
2043
2044 /*
2045  * NAME:        dbAllocBits()
2046  *
2047  * FUNCTION:    allocate a specified block range from a dmap.
2048  *
2049  *              this routine updates the dmap to reflect the working
2050  *              state allocation of the specified block range. it directly
2051  *              updates the bits of the working map and causes the adjustment
2052  *              of the binary buddy system described by the dmap's dmtree
2053  *              leaves to reflect the bits allocated.  it also causes the
2054  *              dmap's dmtree, as a whole, to reflect the allocated range.
2055  *
2056  * PARAMETERS:
2057  *      bmp     −  pointer to bmap descriptor
2058  *      dp      −  pointer to dmap to allocate bits from.
2059  *      blkno   −  starting block number of the bits to be allocated.
2060  *      nblocks −  number of bits to be allocated.
2061  *
2062  * RETURN VALUES: none
2063  *
2064  * serialization: IREAD_LOCK(ipbmap) or IWRITE_LOCK(ipbmap) held on entry/exit;
2065  */
2066 static void dbAllocBits(struct bmap * bmp, struct dmap * dp, s64 blkno,
2067                         int nblocks)
2068 {
2069         int dbitno, word, rembits, nb, nwords, wbitno, nw, agno;
2070         dmtree_t *tp = (dmtree_t *) & dp->tree;
```

```
2071            int size;
2072            s8 *leaf;
2073
2074            /* pick up a pointer to the leaves of the dmap tree */
2075            leaf = dp->tree.stree + LEAFIND;
2076
2077            /* determine the bit number and word within the dmap of the
2078             * starting block.
2079             */
2080            dbitno = blkno & (BPERDMAP - 1);
2081            word = dbitno >> L2DBWORD;
2082
2083            /* block range better be within the dmap */
2084            assert(dbitno + nblocks <= BPERDMAP);
2085
2086            /* allocate the bits of the dmap's words corresponding to the block
2087             * range. not all bits of the first and last words may be contained
2088             * within the block range.  if this is the case, we'll work against
2089             * those words (i.e. partial first and/or last) on an individual basis
2090             * (a single pass), allocating the bits of interest by hand and
2091             * updating the leaf corresponding to the dmap word. a single pass
2092             * will be used for all dmap words fully contained within the
2093             * specified range.  within this pass, the bits of all fully contained
2094             * dmap words will be marked as free in a single shot and the leaves
2095             * will be updated. a single leaf may describe the free space of
2096             * multiple dmap words, so we may update only a subset of the actual
2097             * leaves corresponding to the dmap words of the block range.
2098             */
2099            for (rembits = nblocks; rembits > 0; rembits -= nb, dbitno += nb) {
2100                    /* determine the bit number within the word and
2101                     * the number of bits within the word.
2102                     */
2103                    wbitno = dbitno & (DBWORD - 1);
2104                    nb = min(rembits, DBWORD - wbitno);
2105
2106                    /* check if only part of a word is to be allocated.
2107                     */
2108                    if (nb < DBWORD) {
2109                            /* allocate (set to 1) the appropriate bits within
2110                             * this dmap word.
2111                             */
2112                            dp->wmap[word] |= cpu_to_le32(ONES << (DBWORD - nb)
2113                                                          >> wbitno);
2114
2115                            /* update the leaf for this dmap word. in addition
2116                             * to setting the leaf value to the binary buddy max
2117                             * of the updated dmap word, dbSplit() will split
2118                             * the binary system of the leaves if need be.
2119                             */
2120                            dbSplit(tp, word, BUDMIN,
2121                                    dbMaxBud((u8 *) & dp->wmap[word]));
2122
2123                            word += 1;
2124                    } else {
2125                            /* one or more dmap words are fully contained
2126                             * within the block range.  determine how many
2127                             * words and allocate (set to 1) the bits of these
2128                             * words.
2129                             */
2130                            nwords = rembits >> L2DBWORD;
2131                            memset(&dp->wmap[word], (int) ONES, nwords * 4);
2132
2133                            /* determine how many bits.
2134                             */
2135                            nb = nwords << L2DBWORD;
2136
2137                            /* now update the appropriate leaves to reflect
2138                             * the allocated words.
2139                             */
2140                            for (; nwords > 0; nwords -= nw) {
2141                                    assert(leaf[word] >= BUDMIN);
2142
2143                                    /* determine what the leaf value should be
2144                                     * updated to as the minimum of the l2 number
2145                                     * of bits being allocated and the l2 number
2146                                     * of bits currently described by this leaf.
2147                                     */
2148                                    size = min((int)leaf[word], NLSTOL2BSZ(nwords));
2149
2150                                    /* update the leaf to reflect the allocation.
2151                                     * in addition to setting the leaf value to
2152                                     * NOFREE, dbSplit() will split the binary
2153                                     * system of the leaves to reflect the current
2154                                     * allocation (size).
2155                                     */
2156                                    dbSplit(tp, word, size, NOFREE);
2157
2158                                    /* get the number of dmap words handled */
2159                                    nw = BUDSIZE(size, BUDMIN);
2160                                    word += nw;
```

```
2161                            }
2162                    }
2163            }
2164
2165            /* update the free count for this dmap */
2166            dp->nfree = cpu_to_le32(le32_to_cpu(dp->nfree) − nblocks);
2167
2168            BMAP_LOCK(bmp);
2169
2170            /* if this allocation group is completely free,
2171             * update the maximum allocation group number if this allocation
2172             * group is the new max.
2173             */
2174            agno = blkno >> bmp->db_agl2size;
2175            if (agno > bmp->db_maxag)
2176                    bmp->db_maxag = agno;
2177
2178            /* update the free count for the allocation group and map */
2179            bmp->db_agfree[agno] −= nblocks;
2180            bmp->db_nfree −= nblocks;
2181
2182            BMAP_UNLOCK(bmp);
2183    }
2184
2185
2186    /*
2187     * NAME:        dbFreeBits()
2188     *
2189     * FUNCTION:    free a specified block range from a dmap.
2190     *
2191     *              this routine updates the dmap to reflect the working
2192     *              state allocation of the specified block range. it directly
2193     *              updates the bits of the working map and causes the adjustment
2194     *              of the binary buddy system described by the dmap's dmtree
2195     *              leaves to reflect the bits freed.  it also causes the dmap's
2196     *              dmtree, as a whole, to reflect the deallocated range.
2197     *
2198     * PARAMETERS:
2199     *      bmp     −  pointer to bmap descriptor
2200     *      dp      −  pointer to dmap to free bits from.
2201     *      blkno   −  starting block number of the bits to be freed.
2202     *      nblocks −  number of bits to be freed.
2203     *
2204     * RETURN VALUES: none
2205     *
2206     * serialization: IREAD_LOCK(ipbmap) or IWRITE_LOCK(ipbmap) held on entry/exit;
2207     */
2208    static void dbFreeBits(struct bmap * bmp, struct dmap * dp, s64 blkno,
2209                           int nblocks)
2210    {
2211            int dbitno, word, rembits, nb, nwords, wbitno, nw, agno;
2212            dmtree_t *tp = (dmtree_t *) & dp->tree;
2213            int size;
2214
2215            /* determine the bit number and word within the dmap of the
2216             * starting block.
2217             */
2218            dbitno = blkno & (BPERDMAP − 1);
2219            word = dbitno >> L2DBWORD;
2220
2221            /* block range better be within the dmap.
2222             */
2223            assert(dbitno + nblocks <= BPERDMAP);
2224
2225            /* free the bits of the dmaps words corresponding to the block range.
2226             * not all bits of the first and last words may be contained within
2227             * the block range.  if this is the case, we'll work against those
2228             * words (i.e. partial first and/or last) on an individual basis
2229             * (a single pass), freeing the bits of interest by hand and updating
2230             * the leaf corresponding to the dmap word. a single pass will be used
2231             * for all dmap words fully contained within the specified range.
2232             * within this pass, the bits of all fully contained dmap words will
2233             * be marked as free in a single shot and the leaves will be updated. a
2234             * single leaf may describe the free space of multiple dmap words,
2235             * so we may update only a subset of the actual leaves corresponding
2236             * to the dmap words of the block range.
2237             *
2238             * dbJoin() is used to update leaf values and will join the binary
2239             * buddy system of the leaves if the new leaf values indicate this
2240             * should be done.
2241             */
2242            for (rembits = nblocks; rembits > 0; rembits −= nb, dbitno += nb) {
2243                    /* determine the bit number within the word and
2244                     * the number of bits within the word.
2245                     */
2246                    wbitno = dbitno & (DBWORD − 1);
2247                    nb = min(rembits, DBWORD − wbitno);
2248
2249                    /* check if only part of a word is to be freed.
2250                     */
```

```
2251                    if (nb < DBWORD) {
2252                            /* free (zero) the appropriate bits within this
2253                             * dmap word.
2254                             */
2255                            dp->wmap[word] &=
2256                                cpu_to_le32(~(ONES << (DBWORD - nb)
2257                                              >> wbitno));
2258
2259                            /* update the leaf for this dmap word.
2260                             */
2261                            dbJoin(tp, word,
2262                                   dbMaxBud((u8 *) & dp->wmap[word]));
2263
2264                            word += 1;
2265                    } else {
2266                            /* one or more dmap words are fully contained
2267                             * within the block range.  determine how many
2268                             * words and free (zero) the bits of these words.
2269                             */
2270                            nwords = rembits >> L2DBWORD;
2271                            memset(&dp->wmap[word], 0, nwords * 4);
2272
2273                            /* determine how many bits.
2274                             */
2275                            nb = nwords << L2DBWORD;
2276
2277                            /* now update the appropriate leaves to reflect
2278                             * the freed words.
2279                             */
2280                            for (; nwords > 0; nwords -= nw) {
2281                                    /* determine what the leaf value should be
2282                                     * updated to as the minimum of the l2 number
2283                                     * of bits being freed and the l2 (max) number
2284                                     * of bits that can be described by this leaf.
2285                                     */
2286                                    size =
2287                                        min(LITOL2BSZ
2288                                            (word, L2LPERDMAP, BUDMIN),
2289                                            NLSTOL2BSZ(nwords));
2290
2291                                    /* update the leaf.
2292                                     */
2293                                    dbJoin(tp, word, size);
2294
2295                                    /* get the number of dmap words handled.
2296                                     */
2297                                    nw = BUDSIZE(size, BUDMIN);
2298                                    word += nw;
2299                            }
2300                    }
2301            }
2302
2303            /* update the free count for this dmap.
2304             */
2305            dp->nfree = cpu_to_le32(le32_to_cpu(dp->nfree) + nblocks);
2306
2307            BMAP_LOCK(bmp);
2308
2309            /* update the free count for the allocation group and
2310             * map.
2311             */
2312            agno = blkno >> bmp->db_agl2size;
2313            bmp->db_nfree += nblocks;
2314            bmp->db_agfree[agno] += nblocks;
2315
2316            /* check if this allocation group is not completely free and
2317             * if it is currently the maximum (rightmost) allocation group.
2318             * if so, establish the new maximum allocation group number by
2319             * searching left for the first allocation group with allocation.
2320             */
2321            if ((bmp->db_agfree[agno] == bmp->db_agsize && agno == bmp->db_maxag) ||
2322                (agno == bmp->db_numag - 1 &&
2323                 bmp->db_agfree[agno] == (bmp-> db_mapsize & (BPERDMAP - 1)))) {
2324                    while (bmp->db_maxag > 0) {
2325                            bmp->db_maxag -= 1;
2326                            if (bmp->db_agfree[bmp->db_maxag] !=
2327                                bmp->db_agsize)
2328                                    break;
2329                    }
2330
2331                    /* re-establish the allocation group preference if the
2332                     * current preference is right of the maximum allocation
2333                     * group.
2334                     */
2335                    if (bmp->db_agpref > bmp->db_maxag)
2336                            bmp->db_agpref = bmp->db_maxag;
2337            }
2338
2339            BMAP_UNLOCK(bmp);
2340    }
```

```
2341
2342
2343 /*
2344  * NAME:        dbAdjCtl()
2345  *
2346  * FUNCTION:    adjust a dmap control page at a specified level to reflect
2347  *              the change in a lower level dmap or dmap control page's
2348  *              maximum string of free blocks (i.e. a change in the root
2349  *              of the lower level object's dmtree) due to the allocation
2350  *              or deallocation of a range of blocks with a single dmap.
2351  *
2352  *              on entry, this routine is provided with the new value of
2353  *              the lower level dmap or dmap control page root and the
2354  *              starting block number of the block range whose allocation
2355  *              or deallocation resulted in the root change.  this range
2356  *              is respresented by a single leaf of the current dmapctl
2357  *              and the leaf will be updated with this value, possibly
2358  *              causing a binary buddy system within the leaves to be
2359  *              split or joined.  the update may also cause the dmapctl's
2360  *              dmtree to be updated.
2361  *
2362  *              if the adjustment of the dmap control page, itself, causes its
2363  *              root to change, this change will be bubbled up to the next dmap
2364  *              control level by a recursive call to this routine, specifying
2365  *              the new root value and the next dmap control page level to
2366  *              be adjusted.
2367  * PARAMETERS:
2368  *      bmp     -  pointer to bmap descriptor
2369  *      blkno   -  the first block of a block range within a dmap.  it is
2370  *                 the allocation or deallocation of this block range that
2371  *                 requires the dmap control page to be adjusted.
2372  *      newval  -  the new value of the lower level dmap or dmap control
2373  *                 page root.
2374  *      alloc   -  TRUE if adjustment is due to an allocation.
2375  *      level   -  current level of dmap control page (i.e. L0, L1, L2) to
2376  *                 be adjusted.
2377  *
2378  * RETURN VALUES:
2379  *      0       - success
2380  *      EIO     - i/o error
2381  *
2382  * serialization: IREAD_LOCK(ipbmap) or IWRITE_LOCK(ipbmap) held on entry/exit;
2383  */
2384 static int
2385 dbAdjCtl(struct bmap * bmp, s64 blkno, int newval, int alloc, int level)
2386 {
2387         struct metapage *mp;
2388         s8 oldroot;
2389         int oldval;
2390         s64 lblkno;
2391         struct dmapctl *dcp;
2392         int rc, leafno, ti;
2393
2394         /* get the buffer for the dmap control page for the specified
2395          * block number and control page level.
2396          */
2397         lblkno = BLKTOCTL(blkno, bmp->db_l2nbperpage, level);
2398         mp = read_metapage(bmp->db_ipbmap, lblkno, PSIZE, 0);
2399         if (mp == NULL)
2400                 return (EIO);
2401         dcp = (struct dmapctl *) mp->data;
2402
2403         /* determine the leaf number corresponding to the block and
2404          * the index within the dmap control tree.
2405          */
2406         leafno = BLKTOCTLLEAF(blkno, dcp->budmin);
2407         ti = leafno + le32_to_cpu(dcp->leafidx);
2408
2409         /* save the current leaf value and the current root level (i.e.
2410          * maximum l2 free string described by this dmapctl).
2411          */
2412         oldval = dcp->stree[ti];
2413         oldroot = dcp->stree[ROOT];
2414
2415         /* check if this is a control page update for an allocation.
2416          * if so, update the leaf to reflect the new leaf value using
2417          * dbSplit(); otherwise (deallocation), use dbJoin() to udpate
2418          * the leaf with the new value.  in addition to updating the
2419          * leaf, dbSplit() will also split the binary buddy system of
2420          * the leaves, if required, and bubble new values within the
2421          * dmapctl tree, if required.  similarly, dbJoin() will join
2422          * the binary buddy system of leaves and bubble new values up
2423          * the dmapctl tree as required by the new leaf value.
2424          */
2425         if (alloc) {
2426                 /* check if we are in the middle of a binary buddy
2427                  * system.  this happens when we are performing the
2428                  * first allocation out of an allocation group that
2429                  * is part (not the first part) of a larger binary
2430                  * buddy system.  if we are in the middle, back split
```

```
2431                     * the system prior to calling dbSplit() which assumes
2432                     * that it is at the front of a binary buddy system.
2433                     */
2434                    if (oldval == NOFREE) {
2435                            dbBackSplit((dmtree_t *) dcp, leafno);
2436                            oldval = dcp->stree[ti];
2437                    }
2438                    dbSplit((dmtree_t *) dcp, leafno, dcp->budmin, newval);
2439            } else {
2440                    dbJoin((dmtree_t *) dcp, leafno, newval);
2441            }
2442
2443            /* check if the root of the current dmap control page changed due
2444             * to the update and if the current dmap control page is not at
2445             * the current top level (i.e. L0, L1, L2) of the map.  if so (i.e.
2446             * root changed and this is not the top level), call this routine
2447             * again (recursion) for the next higher level of the mapping to
2448             * reflect the change in root for the current dmap control page.
2449             */
2450            if (dcp->stree[ROOT] != oldroot) {
2451                    /* are we below the top level of the map.  if so,
2452                     * bubble the root up to the next higher level.
2453                     */
2454                    if (level < bmp->db_maxlevel) {
2455                            /* bubble up the new root of this dmap control page to
2456                             * the next level.
2457                             */
2458                            if ((rc =
2459                                 dbAdjCtl(bmp, blkno, dcp->stree[ROOT], alloc,
2460                                          level + 1))) {
2461                                    /* something went wrong in bubbling up the new
2462                                     * root value, so backout the changes to the
2463                                     * current dmap control page.
2464                                     */
2465                                    if (alloc) {
2466                                            dbJoin((dmtree_t *) dcp, leafno,
2467                                                   oldval);
2468                                    } else {
2469                                            /* the dbJoin() above might have
2470                                             * caused a larger binary buddy system
2471                                             * to form and we may now be in the
2472                                             * middle of it.  if this is the case,
2473                                             * back split the buddies.
2474                                             */
2475                                            if (dcp->stree[ti] == NOFREE)
2476                                                    dbBackSplit((dmtree_t *)
2477                                                                dcp, leafno);
2478                                            dbSplit((dmtree_t *) dcp, leafno,
2479                                                    dcp->budmin, oldval);
2480                                    }
2481
2482                                    /* release the buffer and return the error.
2483                                     */
2484                                    release_metapage(mp);
2485                                    return (rc);
2486                            }
2487                    } else {
2488                            /* we're at the top level of the map. update
2489                             * the bmap control page to reflect the size
2490                             * of the maximum free buddy system.
2491                             */
2492                            assert(level == bmp->db_maxlevel);
2493                            assert(bmp->db_maxfreebud == oldroot);
2494                            bmp->db_maxfreebud = dcp->stree[ROOT];
2495                    }
2496            }
2497
2498            /* write the buffer.
2499             */
2500            write_metapage(mp);
2501
2502            return (0);
2503    }
2504
2505
2506    /*
2507     * NAME:        dbSplit()
2508     *
2509     * FUNCTION:    update the leaf of a dmtree with a new value, splitting
2510     *              the leaf from the binary buddy system of the dmtree's
2511     *              leaves, as required.
2512     *
2513     * PARAMETERS:
2514     *      tp      - pointer to the tree containing the leaf.
2515     *      leafno  - the number of the leaf to be updated.
2516     *      splitsz - the size the binary buddy system starting at the leaf
2517     *                must be split to, specified as the log2 number of blocks.
2518     *      newval  - the new value for the leaf.
2519     *
2520     * RETURN VALUES: none
```

```
2521  *
2522  * serialization: IREAD_LOCK(ipbmap) or IWRITE_LOCK(ipbmap) held on entry/exit;
2523  */
2524 static void dbSplit(dmtree_t * tp, int leafno, int splitsz, int newval)
2525 {
2526         int budsz;
2527         int cursz;
2528         s8 *leaf = tp->dmt_stree + le32_to_cpu(tp->dmt_leafidx);
2529
2530         /* check if the leaf needs to be split.
2531          */
2532         if (leaf[leafno] > tp->dmt_budmin) {
2533                 /* the split occurs by cutting the buddy system in half
2534                  * at the specified leaf until we reach the specified
2535                  * size.  pick up the starting split size (current size
2536                  * - 1 in l2) and the corresponding buddy size.
2537                  */
2538                 cursz = leaf[leafno] - 1;
2539                 budsz = BUDSIZE(cursz, tp->dmt_budmin);
2540
2541                 /* split until we reach the specified size.
2542                  */
2543                 while (cursz >= splitsz) {
2544                         /* update the buddy's leaf with its new value.
2545                          */
2546                         dbAdjTree(tp, leafno ^ budsz, cursz);
2547
2548                         /* on to the next size and buddy.
2549                          */
2550                         cursz -= 1;
2551                         budsz >>= 1;
2552                 }
2553         }
2554
2555         /* adjust the dmap tree to reflect the specified leaf's new
2556          * value.
2557          */
2558         dbAdjTree(tp, leafno, newval);
2559 }
2560
2561
2562 /*
2563  * NAME:        dbBackSplit()
2564  *
2565  * FUNCTION:    back split the binary buddy system of dmtree leaves
2566  *              that hold a specified leaf until the specified leaf
2567  *              starts its own binary buddy system.
2568  *
2569  *              the allocators typically perform allocations at the start
2570  *              of binary buddy systems and dbSplit() is used to accomplish
2571  *              any required splits.  in some cases, however, allocation
2572  *              may occur in the middle of a binary system and requires a
2573  *              back split, with the split proceeding out from the middle of
2574  *              the system (less efficient) rather than the start of the
2575  *              system (more efficient).  the cases in which a back split
2576  *              is required are rare and are limited to the first allocation
2577  *              within an allocation group which is a part (not first part)
2578  *              of a larger binary buddy system and a few exception cases
2579  *              in which a previous join operation must be backed out.
2580  *
2581  * PARAMETERS:
2582  *      tp      - pointer to the tree containing the leaf.
2583  *      leafno  - the number of the leaf to be updated.
2584  *
2585  * RETURN VALUES: none
2586  *
2587  * serialization: IREAD_LOCK(ipbmap) or IWRITE_LOCK(ipbmap) held on entry/exit;
2588  */
2589 static void dbBackSplit(dmtree_t * tp, int leafno)
2590 {
2591         int budsz, bud, w, bsz, size;
2592         int cursz;
2593         s8 *leaf = tp->dmt_stree + le32_to_cpu(tp->dmt_leafidx);
2594
2595         /* leaf should be part (not first part) of a binary
2596          * buddy system.
2597          */
2598         assert(leaf[leafno] == NOFREE);
2599
2600         /* the back split is accomplished by iteratively finding the leaf
2601          * that starts the buddy system that contains the specified leaf and
2602          * splitting that system in two.  this iteration continues until
2603          * the specified leaf becomes the start of a buddy system.
2604          *
2605          * determine maximum possible l2 size for the specified leaf.
2606          */
2607         size =
2608             LITOL2BSZ(leafno, le32_to_cpu(tp->dmt_l2nleafs),
2609                       tp->dmt_budmin);
2610
```

```
2611          /* determine the number of leaves covered by this size.  this
2612           * is the buddy size that we will start with as we search for
2613           * the buddy system that contains the specified leaf.
2614           */
2615          budsz = BUDSIZE(size, tp->dmt_budmin);
2616
2617          /* back split.
2618           */
2619          while (leaf[leafno] == NOFREE) {
2620                  /* find the leftmost buddy leaf.
2621                   */
2622                  for (w = leafno, bsz = budsz;; bsz <<= 1,
2623                       w = (w < bud) ? w : bud) {
2624                     assert(bsz < le32_to_cpu(tp->dmt_nleafs));
2625
2626                     /* determine the buddy.
2627                      */
2628                     bud = w ^ bsz;
2629
2630                     /* check if this buddy is the start of the system.
2631                      */
2632                     if (leaf[bud] != NOFREE) {
2633                             /* split the leaf at the start of the
2634                              * system in two.
2635                              */
2636                             cursz = leaf[bud] - 1;
2637                             dbSplit(tp, bud, cursz, cursz);
2638                             break;
2639                     }
2640                  }
2641          }
2642
2643          assert(leaf[leafno] == size);
2644  }
2645
2646
2647  /*
2648   * NAME:        dbJoin()
2649   *
2650   * FUNCTION:    update the leaf of a dmtree with a new value, joining
2651   *              the leaf with other leaves of the dmtree into a multi-leaf
2652   *              binary buddy system, as required.
2653   *
2654   * PARAMETERS:
2655   *      tp      - pointer to the tree containing the leaf.
2656   *      leafno  - the number of the leaf to be updated.
2657   *      newval  - the new value for the leaf.
2658   *
2659   * RETURN VALUES: none
2660   */
2661  static void dbJoin(dmtree_t * tp, int leafno, int newval)
2662  {
2663          int budsz, buddy;
2664          s8 *leaf;
2665
2666          /* can the new leaf value require a join with other leaves ?
2667           */
2668          if (newval >= tp->dmt_budmin) {
2669                  /* pickup a pointer to the leaves of the tree.
2670                   */
2671                  leaf = tp->dmt_stree + le32_to_cpu(tp->dmt_leafidx);
2672
2673                  /* try to join the specified leaf into a large binary
2674                   * buddy system.  the join proceeds by attempting to join
2675                   * the specified leafno with its buddy (leaf) at new value.
2676                   * if the join occurs, we attempt to join the left leaf
2677                   * of the joined buddies with its buddy at new value + 1.
2678                   * we continue to join until we find a buddy that cannot be
2679                   * joined (does not have a value equal to the size of the
2680                   * last join) or until all leaves have been joined into a
2681                   * single system.
2682                   *
2683                   * get the buddy size (number of words covered) of
2684                   * the new value.
2685                   */
2686                  budsz = BUDSIZE(newval, tp->dmt_budmin);
2687
2688                  /* try to join.
2689                   */
2690                  while (budsz < le32_to_cpu(tp->dmt_nleafs)) {
2691                          /* get the buddy leaf.
2692                           */
2693                          buddy = leafno ^ budsz;
2694
2695                          /* if the leaf's new value is greater than its
2696                           * buddy's value, we join no more.
2697                           */
2698                          if (newval > leaf[buddy])
2699                                  break;
2700
```

```
2701                            assert(newval == leaf[buddy]);
2702
2703                            /* check which (leafno or buddy) is the left buddy.
2704                             * the left buddy gets to claim the blocks resulting
2705                             * from the join while the right gets to claim none.
2706                             * the left buddy is also eligable to participate in
2707                             * a join at the next higher level while the right
2708                             * is not.
2709                             *
2710                             */
2711                            if (leafno < buddy) {
2712                                    /* leafno is the left buddy.
2713                                     */
2714                                    dbAdjTree(tp, buddy, NOFREE);
2715                            } else {
2716                                    /* buddy is the left buddy and becomes
2717                                     * leafno.
2718                                     */
2719                                    dbAdjTree(tp, leafno, NOFREE);
2720                                    leafno = buddy;
2721                            }
2722
2723                            /* on to try the next join.
2724                             */
2725                            newval += 1;
2726                            budsz <<= 1;
2727                    }
2728            }
2729
2730            /* update the leaf value.
2731             */
2732            dbAdjTree(tp, leafno, newval);
2733    }
2734
2735
2736    /*
2737     * NAME:        dbAdjTree()
2738     *
2739     * FUNCTION:    update a leaf of a dmtree with a new value, adjusting
2740     *              the dmtree, as required, to reflect the new leaf value.
2741     *              the combination of any buddies must already be done before
2742     *              this is called.
2743     *
2744     * PARAMETERS:
2745     *      tp      - pointer to the tree to be adjusted.
2746     *      leafno  - the number of the leaf to be updated.
2747     *      newval  - the new value for the leaf.
2748     *
2749     * RETURN VALUES: none
2750     */
2751    static void dbAdjTree(dmtree_t * tp, int leafno, int newval)
2752    {
2753            int lp, pp, k;
2754            int max;
2755
2756            /* pick up the index of the leaf for this leafno.
2757             */
2758            lp = leafno + le32_to_cpu(tp->dmt_leafidx);
2759
2760            /* is the current value the same as the old value ?  if so,
2761             * there is nothing to do.
2762             */
2763            if (tp->dmt_stree[lp] == newval)
2764                    return;
2765
2766            /* set the new value.
2767             */
2768            tp->dmt_stree[lp] = newval;
2769
2770            /* bubble the new value up the tree as required.
2771             */
2772            for (k = 0; k < le32_to_cpu(tp->dmt_height); k++) {
2773                    /* get the index of the first leaf of the 4 leaf
2774                     * group containing the specified leaf (leafno).
2775                     */
2776                    lp = ((lp - 1) & ~0x03) + 1;
2777
2778                    /* get the index of the parent of this 4 leaf group.
2779                     */
2780                    pp = (lp - 1) >> 2;
2781
2782                    /* determine the maximum of the 4 leaves.
2783                     */
2784                    max = TREEMAX(&tp->dmt_stree[lp]);
2785
2786                    /* if the maximum of the 4 is the same as the
2787                     * parent's value, we're done.
2788                     */
2789                    if (tp->dmt_stree[pp] == max)
2790                            break;
```

```
2791
2792                    /* parent gets new value.
2793                     */
2794                    tp->dmt_stree[pp] = max;
2795
2796                    /* parent becomes leaf for next go-round.
2797                     */
2798                    lp = pp;
2799            }
2800    }
2801
2802
2803    /*
2804     * NAME:        dbFindLeaf()
2805     *
2806     * FUNCTION:    search a dmtree_t for sufficient free blocks, returning
2807     *              the index of a leaf describing the free blocks if
2808     *              sufficient free blocks are found.
2809     *
2810     *              the search starts at the top of the dmtree_t tree and
2811     *              proceeds down the tree to the leftmost leaf with sufficient
2812     *              free space.
2813     *
2814     * PARAMETERS:
2815     *      tp      - pointer to the tree to be searched.
2816     *      l2nb    - log2 number of free blocks to search for.
2817     *      leafidx - return pointer to be set to the index of the leaf
2818     *                describing at least l2nb free blocks if sufficient
2819     *                free blocks are found.
2820     *
2821     * RETURN VALUES:
2822     *      0       - success
2823     *      ENOSPC  - insufficient free blocks.
2824     */
2825    static int dbFindLeaf(dmtree_t * tp, int l2nb, int *leafidx)
2826    {
2827            int ti, n = 0, k, x = 0;
2828
2829            /* first check the root of the tree to see if there is
2830             * sufficient free space.
2831             */
2832            if (l2nb > tp->dmt_stree[ROOT])
2833                    return (ENOSPC);
2834
2835            /* sufficient free space available. now search down the tree
2836             * starting at the next level for the leftmost leaf that
2837             * describes sufficient free space.
2838             */
2839            for (k = le32_to_cpu(tp->dmt_height), ti = 1;
2840                 k > 0; k--, ti = ((ti + n) << 2) + 1) {
2841                    /* search the four nodes at this level, starting from
2842                     * the left.
2843                     */
2844                    for (x = ti, n = 0; n < 4; n++) {
2845                            /* sufficient free space found.  move to the next
2846                             * level (or quit if this is the last level).
2847                             */
2848                            if (l2nb <= tp->dmt_stree[x + n])
2849                                    break;
2850                    }
2851
2852                    /* better have found something since the higher
2853                     * levels of the tree said it was here.
2854                     */
2855                    assert(n < 4);
2856            }
2857
2858            /* set the return to the leftmost leaf describing sufficient
2859             * free space.
2860             */
2861            *leafidx = x + n - le32_to_cpu(tp->dmt_leafidx);
2862
2863            return (0);
2864    }
2865
2866
2867    /*
2868     * NAME:        dbFindBits()
2869     *
2870     * FUNCTION:    find a specified number of binary buddy free bits within a
2871     *              dmap bitmap word value.
2872     *
2873     *              this routine searches the bitmap value for (1 << l2nb) free
2874     *              bits at (1 << l2nb) alignments within the value.
2875     *
2876     * PARAMETERS:
2877     *      word    -  dmap bitmap word value.
2878     *      l2nb    -  number of free bits specified as a log2 number.
2879     *
2880     * RETURN VALUES:
```

```
2881  *      starting bit number of free bits.
2882  */
2883 static int dbFindBits(u32 word, int l2nb)
2884 {
2885         int bitno, nb;
2886         u32 mask;
2887
2888         /* get the number of bits.
2889          */
2890         nb = 1 << l2nb;
2891         assert(nb <= DBWORD);
2892
2893         /* complement the word so we can use a mask (i.e. 0s represent
2894          * free bits) and compute the mask.
2895          */
2896         word = ~word;
2897         mask = ONES << (DBWORD − nb);
2898
2899         /* scan the word for nb free bits at nb alignments.
2900          */
2901         for (bitno = 0; mask != 0; bitno += nb, mask >>= nb) {
2902                 if ((mask & word) == mask)
2903                         break;
2904         }
2905
2906         ASSERT(bitno < 32);
2907
2908         /* return the bit number.
2909          */
2910         return (bitno);
2911 }
2912
2913
2914 /*
2915  * NAME:        dbMaxBud(u8 *cp)
2916  *
2917  * FUNCTION:    determine the largest binary buddy string of free
2918  *              bits within 32−bits of the map.
2919  *
2920  * PARAMETERS:
2921  *      cp      −  pointer to the 32−bit value.
2922  *
2923  * RETURN VALUES:
2924  *      largest binary buddy of free bits within a dmap word.
2925  */
2926 static int dbMaxBud(u8 * cp)
2927 {
2928         signed char tmp1, tmp2;
2929
2930         /* check if the wmap word is all free. if so, the
2931          * free buddy size is BUDMIN.
2932          */
2933         if (*((uint *) cp) == 0)
2934                 return (BUDMIN);
2935
2936         /* check if the wmap word is half free. if so, the
2937          * free buddy size is BUDMIN−1.
2938          */
2939         if (*((u16 *) cp) == 0 || *((u16 *) cp + 1) == 0)
2940                 return (BUDMIN − 1);
2941
2942         /* not all free or half free. determine the free buddy
2943          * size thru table lookup using quarters of the wmap word.
2944          */
2945         tmp1 = max(budtab[cp[2]], budtab[cp[3]]);
2946         tmp2 = max(budtab[cp[0]], budtab[cp[1]]);
2947         return (max(tmp1, tmp2));
2948 }
2949
2950
2951 /*
2952  * NAME:        cnttz(uint word)
2953  *
2954  * FUNCTION:    determine the number of trailing zeros within a 32−bit
2955  *              value.
2956  *
2957  * PARAMETERS:
2958  *      value   −  32−bit value to be examined.
2959  *
2960  * RETURN VALUES:
2961  *      count of trailing zeros
2962  */
2963 int cnttz(u32 word)
2964 {
2965         int n;
2966
2967         for (n = 0; n < 32; n++, word >>= 1) {
2968                 if (word & 0x01)
2969                         break;
2970         }
```

```
2971
2972            return (n);
2973    }
2974
2975
2976    /*
2977     * NAME:        cntlz(u32 value)
2978     *
2979     * FUNCTION:    determine the number of leading zeros within a 32−bit
2980     *              value.
2981     *
2982     * PARAMETERS:
2983     *      value   −  32−bit value to be examined.
2984     *
2985     * RETURN VALUES:
2986     *      count of leading zeros
2987     */
2988    int cntlz(u32 value)
2989    {
2990            int n;
2991
2992            for (n = 0; n < 32; n++, value <<= 1) {
2993                    if (value & HIGHORDER)
2994                            break;
2995            }
2996            return (n);
2997    }
2998
2999
3000    /*
3001     * NAME:        blkstol2(s64 nb)
3002     *
3003     * FUNCTION:    convert a block count to its log2 value. if the block
3004     *              count is not a l2 multiple, it is rounded up to the next
3005     *              larger l2 multiple.
3006     *
3007     * PARAMETERS:
3008     *      nb      −  number of blocks
3009     *
3010     * RETURN VALUES:
3011     *      log2 number of blocks
3012     */
3013    int blkstol2(s64 nb)
3014    {
3015            int l2nb;
3016            s64 mask;               /* meant to be signed */
3017
3018            mask = (s64) 1 << (64 − 1);
3019
3020            /* count the leading bits.
3021             */
3022            for (l2nb = 0; l2nb < 64; l2nb++, mask >>= 1) {
3023                    /* leading bit found.
3024                     */
3025                    if (nb & mask) {
3026                            /* determine the l2 value.
3027                             */
3028                            l2nb = (64 − 1) − l2nb;
3029
3030                            /* check if we need to round up.
3031                             */
3032                            if (~mask & nb)
3033                                    l2nb++;
3034
3035                            return (l2nb);
3036                    }
3037            }
3038            assert(0);
3039            return 0;               /* fix compiler warning */
3040    }
3041
3042
3043    /*
3044     * NAME:        fsDirty()
3045     *
3046     * FUNCTION:    xxx
3047     *
3048     * PARAMETERS:
3049     *      ipmnt   − mount inode
3050     *
3051     * RETURN VALUES:
3052     *      none
3053     */
3054    void fsDirty()
3055    {
3056            printk("fsDirty(): bye−bye\n");
3057            assert(0);
3058    }
3059
3060
```

```
3061 /*
3062  * NAME:        dbAllocBottomUp()
3063  *
3064  * FUNCTION:    alloc the specified block range from the working block
3065  *              allocation map.
3066  *
3067  *              the blocks will be alloc from the working map one dmap
3068  *              at a time.
3069  *
3070  * PARAMETERS:
3071  *      ip      −  pointer to in−core inode;
3072  *      blkno   −  starting block number to be freed.
3073  *      nblocks −  number of blocks to be freed.
3074  *
3075  * RETURN VALUES:
3076  *      0       − success
3077  *      EIO     − i/o error
3078  */
3079 int dbAllocBottomUp(struct inode *ip, s64 blkno, s64 nblocks)
3080 {
3081         struct metapage *mp;
3082         struct dmap *dp;
3083         int nb, rc;
3084         s64 lblkno, rem;
3085         struct inode *ipbmap = JFS_SBI(ip−>i_sb)−>ipbmap;
3086         struct bmap *bmp = JFS_SBI(ip−>i_sb)−>bmap;
3087
3088         IREAD_LOCK(ipbmap);
3089
3090         /* block to be allocated better be within the mapsize. */
3091         ASSERT(nblocks <= bmp−>db_mapsize − blkno);
3092
3093         /*
3094          * allocate the blocks a dmap at a time.
3095          */
3096         mp = NULL;
3097         for (rem = nblocks; rem > 0; rem −= nb, blkno += nb) {
3098                 /* release previous dmap if any */
3099                 if (mp) {
3100                         write_metapage(mp);
3101                 }
3102
3103                 /* get the buffer for the current dmap. */
3104                 lblkno = BLKTODMAP(blkno, bmp−>db_l2nbperpage);
3105                 mp = read_metapage(ipbmap, lblkno, PSIZE, 0);
3106                 if (mp == NULL) {
3107                         IREAD_UNLOCK(ipbmap);
3108                         return (EIO);
3109                 }
3110                 dp = (struct dmap *) mp−>data;
3111
3112                 /* determine the number of blocks to be allocated from
3113                  * this dmap.
3114                  */
3115                 nb = min(rem, BPERDMAP − (blkno & (BPERDMAP − 1)));
3116
3117                 DBFREECK(bmp−>db_DBmap, bmp−>db_mapsize, blkno, nb);
3118
3119                 /* allocate the blocks. */
3120                 if ((rc = dbAllocDmapBU(bmp, dp, blkno, nb))) {
3121                         release_metapage(mp);
3122                         IREAD_UNLOCK(ipbmap);
3123                         return (rc);
3124                 }
3125
3126                 DBALLOC(bmp−>db_DBmap, bmp−>db_mapsize, blkno, nb);
3127         }
3128
3129         /* write the last buffer. */
3130         write_metapage(mp);
3131
3132         IREAD_UNLOCK(ipbmap);
3133
3134         return (0);
3135 }
3136
3137
3138 static int dbAllocDmapBU(struct bmap * bmp, struct dmap * dp, s64 blkno,
3139                          int nblocks)
3140 {
3141         int rc;
3142         int dbitno, word, rembits, nb, nwords, wbitno, agno;
3143         s8 oldroot, *leaf;
3144         struct dmaptree *tp = (struct dmaptree *) & dp−>tree;
3145
3146         /* save the current value of the root (i.e. maximum free string)
3147          * of the dmap tree.
3148          */
3149         oldroot = tp−>stree[ROOT];
3150
```

```
3151          /* pick up a pointer to the leaves of the dmap tree */
3152          leaf = tp->stree + LEAFIND;
3153
3154          /* determine the bit number and word within the dmap of the
3155           * starting block.
3156           */
3157          dbitno = blkno & (BPERDMAP − 1);
3158          word = dbitno >> L2DBWORD;
3159
3160          /* block range better be within the dmap */
3161          assert(dbitno + nblocks <= BPERDMAP);
3162
3163          /* allocate the bits of the dmap's words corresponding to the block
3164           * range. not all bits of the first and last words may be contained
3165           * within the block range.  if this is the case, we'll work against
3166           * those words (i.e. partial first and/or last) on an individual basis
3167           * (a single pass), allocating the bits of interest by hand and
3168           * updating the leaf corresponding to the dmap word. a single pass
3169           * will be used for all dmap words fully contained within the
3170           * specified range.  within this pass, the bits of all fully contained
3171           * dmap words will be marked as free in a single shot and the leaves
3172           * will be updated. a single leaf may describe the free space of
3173           * multiple dmap words, so we may update only a subset of the actual
3174           * leaves corresponding to the dmap words of the block range.
3175           */
3176          for (rembits = nblocks; rembits > 0; rembits −= nb, dbitno += nb) {
3177                  /* determine the bit number within the word and
3178                   * the number of bits within the word.
3179                   */
3180                  wbitno = dbitno & (DBWORD − 1);
3181                  nb = min(rembits, DBWORD − wbitno);
3182
3183                  /* check if only part of a word is to be allocated.
3184                   */
3185                  if (nb < DBWORD) {
3186                          /* allocate (set to 1) the appropriate bits within
3187                           * this dmap word.
3188                           */
3189                          dp->wmap[word] |= cpu_to_le32(ONES << (DBWORD − nb)
3190                                                        >> wbitno);
3191
3192                          word++;
3193                  } else {
3194                          /* one or more dmap words are fully contained
3195                           * within the block range.  determine how many
3196                           * words and allocate (set to 1) the bits of these
3197                           * words.
3198                           */
3199                          nwords = rembits >> L2DBWORD;
3200                          memset(&dp->wmap[word], (int) ONES, nwords * 4);
3201
3202                          /* determine how many bits */
3203                          nb = nwords << L2DBWORD;
3204                          word += nwords;
3205                  }
3206          }
3207
3208          /* update the free count for this dmap */
3209          dp->nfree = cpu_to_le32(le32_to_cpu(dp->nfree) − nblocks);
3210
3211          /* reconstruct summary tree */
3212          dbInitDmapTree(dp);
3213
3214          BMAP_LOCK(bmp);
3215
3216          /* if this allocation group is completely free,
3217           * update the highest active allocation group number
3218           * if this allocation group is the new max.
3219           */
3220          agno = blkno >> bmp->db_agl2size;
3221          if (agno > bmp->db_maxag)
3222                  bmp->db_maxag = agno;
3223
3224          /* update the free count for the allocation group and map */
3225          bmp->db_agfree[agno] −= nblocks;
3226          bmp->db_nfree −= nblocks;
3227
3228          BMAP_UNLOCK(bmp);
3229
3230          /* if the root has not changed, done. */
3231          if (tp->stree[ROOT] == oldroot)
3232                  return (0);
3233
3234          /* root changed. bubble the change up to the dmap control pages.
3235           * if the adjustment of the upper level control pages fails,
3236           * backout the bit allocation (thus making everything consistent).
3237           */
3238          if ((rc = dbAdjCtl(bmp, blkno, tp->stree[ROOT], 1, 0)))
3239                  dbFreeBits(bmp, dp, blkno, nblocks);
3240
```

```
3241            return (rc);
3242    }
3243
3244
3245    /*
3246     * NAME:        dbExtendFS()
3247     *
3248     * FUNCTION:    extend bmap from blkno for nblocks;
3249     *              dbExtendFS() updates bmap ready for dbAllocBottomUp();
3250     *
3251     * L2
3252     *  |
3253     *   L1----------------------------------L1
3254     *    |                                  |
3255     *     L0---------L0---------L0           L0---------L0---------L0
3256     *      |          |          |            |          |          |
3257     *       d0,...,dn  d0,...,dn  d0,...,dn     d0,...,dn  d0,...,dn  d0,.,dm;
3258     * L2L1L0d0,...,dnL0d0,...,dnL0d0,...,dnL1L0d0,...,dnL0d0,...,dnL0d0,..dm
3259     *
3260     * <---old---><----------------------------extend---------------------->
3261     */
3262    int dbExtendFS(struct inode *ipbmap, s64 blkno, s64 nblocks)
3263    {
3264            struct jfs_sb_info *sbi = JFS_SBI(ipbmap->i_sb);
3265            int nbperpage = sbi->nbperpage;
3266            int i, i0 = TRUE, j, j0 = TRUE, k, n;
3267            s64 newsize;
3268            s64 p;
3269            struct metapage *mp, *l2mp, *l1mp, *l0mp;
3270            struct dmapctl *l2dcp, *l1dcp, *l0dcp;
3271            struct dmap *dp;
3272            s8 *l0leaf, *l1leaf, *l2leaf;
3273            struct bmap *bmp = sbi->bmap;
3274            int agno, l2agsize, oldl2agsize;
3275            s64 ag_rem;
3276
3277            newsize = blkno + nblocks;
3278
3279            jEVENT(0, ("dbExtendFS: blkno:%Ld nblocks:%Ld newsize:%Ld\n",
3280                       (long long) blkno, (long long) nblocks,
3281                       (long long) newsize));
3282
3283            /*
3284             *      initialize bmap control page.
3285             *
3286             * all the data in bmap control page should exclude
3287             * the mkfs hidden dmap page.
3288             */
3289
3290            /* update mapsize */
3291            bmp->db_mapsize = newsize;
3292            bmp->db_maxlevel = BMAPSZTOLEV(bmp->db_mapsize);
3293
3294            /* compute new AG size */
3295            l2agsize = dbGetL2AGSize(newsize);
3296            oldl2agsize = bmp->db_agl2size;
3297
3298            bmp->db_agl2size = l2agsize;
3299            bmp->db_agsize = 1 << l2agsize;
3300
3301            /* compute new number of AG */
3302            agno = bmp->db_numag;
3303            bmp->db_numag = newsize >> l2agsize;
3304            bmp->db_numag += ((u32) newsize % (u32) bmp->db_agsize) ? 1 : 0;
3305
3306            /*
3307             *      reconfigure db_agfree[]
3308             * from old AG configuration to new AG configuration;
3309             *
3310             * coalesce contiguous k (newAGSize/oldAGSize) AGs;
3311             * i.e., (AGi, ..., AGj) where i = k*n and j = k*(n+1) - 1 to AGn;
3312             * note: new AG size = old AG size * (2**x).
3313             */
3314            if (l2agsize == oldl2agsize)
3315                    goto extend;
3316            k = 1 << (l2agsize - oldl2agsize);
3317            ag_rem = bmp->db_agfree[0];     /* save agfree[0] */
3318            for (i = 0, n = 0; i < agno; n++) {
3319                    bmp->db_agfree[n] = 0;  /* init collection point */
3320
3321                    /* coalesce cotiguous k AGs; */
3322                    for (j = 0; j < k && i < agno; j++, i++) {
3323                            /* merge AGi to AGn */
3324                            bmp->db_agfree[n] += bmp->db_agfree[i];
3325                    }
3326            }
3327            bmp->db_agfree[0] += ag_rem;    /* restore agfree[0] */
3328
3329            for (; n < MAXAG; n++)
3330                    bmp->db_agfree[n] = 0;
```

```
3331
3332            /*
3333             * update highest active ag number
3334             */
3335
3336            bmp->db_maxag = bmp->db_maxag / k;
3337
3338            /*
3339             *      extend bmap
3340             *
3341             * update bit maps and corresponding level control pages;
3342             * global control page db_nfree, db_agfree[agno], db_maxfreebud;
3343             */
3344           extend:
3345            /* get L2 page */
3346            p = BMAPBLKNO + nbperpage;      /* L2 page */
3347            l2mp = read_metapage(ipbmap, p, PSIZE, 0);
3348            assert(l2mp);
3349            l2dcp = (struct dmapctl *) l2mp->data;
3350
3351            /* compute start L1 */
3352            k = blkno >> L2MAXL1SIZE;
3353            l2leaf = l2dcp->stree + CTLLEAFIND + k;
3354            p = BLKTOL1(blkno, sbi->l2nbperpage);   /* L1 page */
3355
3356            /*
3357             * extend each L1 in L2
3358             */
3359            for (; k < LPERCTL; k++, p += nbperpage) {
3360                    /* get L1 page */
3361                    if (j0) {
3362                            /* read in L1 page: (blkno & (MAXL1SIZE - 1)) */
3363                            l1mp = read_metapage(ipbmap, p, PSIZE, 0);
3364                            if (l1mp == NULL)
3365                                    goto errout;
3366                            l1dcp = (struct dmapctl *) l1mp->data;
3367
3368                            /* compute start L0 */
3369                            j = (blkno & (MAXL1SIZE - 1)) >> L2MAXL0SIZE;
3370                            l1leaf = l1dcp->stree + CTLLEAFIND + j;
3371                            p = BLKTOL0(blkno, sbi->l2nbperpage);
3372                            j0 = FALSE;
3373                    } else {
3374                            /* assign/init L1 page */
3375                            l1mp = get_metapage(ipbmap, p, PSIZE, 0);
3376                            if (l1mp == NULL)
3377                                    goto errout;
3378
3379                            l1dcp = (struct dmapctl *) l1mp->data;
3380
3381                            /* compute start L0 */
3382                            j = 0;
3383                            l1leaf = l1dcp->stree + CTLLEAFIND;
3384                            p += nbperpage; /* 1st L0 of L1.k  */
3385                    }
3386
3387                    /*
3388                     * extend each L0 in L1
3389                     */
3390                    for (; j < LPERCTL; j++) {
3391                            /* get L0 page */
3392                            if (i0) {
3393                                    /* read in L0 page: (blkno & (MAXL0SIZE - 1)) */
3394
3395                                    l0mp = read_metapage(ipbmap, p, PSIZE, 0);
3396                                    if (l0mp == NULL)
3397                                            goto errout;
3398                                    l0dcp = (struct dmapctl *) l0mp->data;
3399
3400                                    /* compute start dmap */
3401                                    i = (blkno & (MAXL0SIZE - 1)) >>
3402                                        L2BPERDMAP;
3403                                    l0leaf = l0dcp->stree + CTLLEAFIND + i;
3404                                    p = BLKTODMAP(blkno,
3405                                                  sbi->l2nbperpage);
3406                                    i0 = FALSE;
3407                            } else {
3408                                    /* assign/init L0 page */
3409                                    l0mp = get_metapage(ipbmap, p, PSIZE, 0);
3410                                    if (l0mp == NULL)
3411                                            goto errout;
3412
3413                                    l0dcp = (struct dmapctl *) l0mp->data;
3414
3415                                    /* compute start dmap */
3416                                    i = 0;
3417                                    l0leaf = l0dcp->stree + CTLLEAFIND;
3418                                    p += nbperpage; /* 1st dmap of L0.j */
3419                            }
3420
```

```
3421                        /*
3422                         * extend each dmap in L0
3423                         */
3424                        for (; i < LPERCTL; i++) {
3425                                /*
3426                                 * reconstruct the dmap page, and
3427                                 * initialize corresponding parent L0 leaf
3428                                 */
3429                                if ((n = blkno & (BPERDMAP − 1))) {
3430                                        /* read in dmap page: */
3431                                        mp = read_metapage(ipbmap, p,
3432                                                           PSIZE, 0);
3433                                        if (mp == NULL)
3434                                                goto errout;
3435                                        n = min(nblocks, (s64)BPERDMAP − n);
3436                                } else {
3437                                        /* assign/init dmap page */
3438                                        mp = read_metapage(ipbmap, p,
3439                                                           PSIZE, 0);
3440                                        if (mp == NULL)
3441                                                goto errout;
3442
3443                                        n = min(nblocks, (s64)BPERDMAP);
3444                                }
3445
3446                                dp = (struct dmap *) mp->data;
3447                                *l0leaf = dbInitDmap(dp, blkno, n);
3448
3449                                bmp->db_nfree += n;
3450                                agno = le64_to_cpu(dp->start) >> l2agsize;
3451                                bmp->db_agfree[agno] += n;
3452
3453                                write_metapage(mp);
3454
3455                                l0leaf++;
3456                                p += nbperpage;
3457
3458                                blkno += n;
3459                                nblocks −= n;
3460                                if (nblocks == 0)
3461                                        break;
3462                        }       /* for each dmap in a L0 */
3463
3464                        /*
3465                         * build current L0 page from its leaves, and
3466                         * initialize corresponding parent L1 leaf
3467                         */
3468                        *l1leaf = dbInitDmapCtl(l0dcp, 0, ++i);
3469                        write_metapage(l0mp);
3470
3471                        if (nblocks)
3472                                l1leaf++;       /* continue for next L0 */
3473                        else {
3474                                /* more than 1 L0 ? */
3475                                if (j > 0)
3476                                        break;  /* build L1 page */
3477                                else {
3478                                        /* summarize in global bmap page */
3479                                        bmp->db_maxfreebud = *l1leaf;
3480                                        release_metapage(l1mp);
3481                                        release_metapage(l2mp);
3482                                        goto finalize;
3483                                }
3484                        }
3485                }               /* for each L0 in a L1 */
3486
3487                /*
3488                 * build current L1 page from its leaves, and
3489                 * initialize corresponding parent L2 leaf
3490                 */
3491                *l2leaf = dbInitDmapCtl(l1dcp, 1, ++j);
3492                write_metapage(l1mp);
3493
3494                if (nblocks)
3495                        l2leaf++;       /* continue for next L1 */
3496                else {
3497                        /* more than 1 L1 ? */
3498                        if (k > 0)
3499                                break;  /* build L2 page */
3500                        else {
3501                                /* summarize in global bmap page */
3502                                bmp->db_maxfreebud = *l2leaf;
3503                                release_metapage(l2mp);
3504                                goto finalize;
3505                        }
3506                }
3507        }                       /* for each L1 in a L2 */
3508
3509        assert(0);
3510
```

```
3511          /*
3512           *      finalize bmap control page
3513           */
3514        finalize:
3515
3516          return 0;
3517
3518        errout:
3519          return EIO;
3520  }
3521
3522
3523  /*
3524   *      dbFinalizeBmap()
3525   */
3526  void dbFinalizeBmap(struct inode *ipbmap)
3527  {
3528          struct bmap *bmp = JFS_SBI(ipbmap->i_sb)->bmap;
3529          int actags, inactags, l2nl;
3530          s64 ag_rem, actfree, inactfree, avgfree;
3531          int i, n;
3532
3533          /*
3534           *      finalize bmap control page
3535           */
3536  //finalize:
3537          /*
3538           * compute db_agpref: preferred ag to allocate from
3539           * (the leftmost ag with average free space in it);
3540           */
3541  //agpref:
3542          /* get the number of active ags and inacitve ags */
3543          actags = bmp->db_maxag + 1;
3544          inactags = bmp->db_numag - actags;
3545          ag_rem = bmp->db_mapsize & (bmp->db_agsize - 1);        /* ??? */
3546
3547          /* determine how many blocks are in the inactive allocation
3548           * groups. in doing this, we must account for the fact that
3549           * the rightmost group might be a partial group (i.e. file
3550           * system size is not a multiple of the group size).
3551           */
3552          inactfree = (inactags && ag_rem) ?
3553              ((inactags - 1) << bmp->db_agl2size) + ag_rem
3554              : inactags << bmp->db_agl2size;
3555
3556          /* determine how many free blocks are in the active
3557           * allocation groups plus the average number of free blocks
3558           * within the active ags.
3559           */
3560          actfree = bmp->db_nfree - inactfree;
3561          avgfree = (u32) actfree / (u32) actags;
3562
3563          /* if the preferred allocation group has not average free space.
3564           * re-establish the preferred group as the leftmost
3565           * group with average free space.
3566           */
3567          if (bmp->db_agfree[bmp->db_agpref] < avgfree) {
3568                  for (bmp->db_agpref = 0; bmp->db_agpref < actags;
3569                       bmp->db_agpref++) {
3570                          if (bmp->db_agfree[bmp->db_agpref] >= avgfree)
3571                                  break;
3572                  }
3573                  assert(bmp->db_agpref < bmp->db_numag);
3574          }
3575
3576          /*
3577           * compute db_aglevel, db_agheigth, db_width, db_agstart:
3578           * an ag is covered in aglevel dmapctl summary tree,
3579           * at agheight level height (from leaf) with agwidth number of nodes
3580           * each, which starts at agstart index node of the smmary tree node
3581           * array;
3582           */
3583          bmp->db_aglevel = BMAPSZTOLEV(bmp->db_agsize);
3584          l2nl =
3585              bmp->db_agl2size - (L2BPERDMAP + bmp->db_aglevel * L2LPERCTL);
3586          bmp->db_agheigth = l2nl >> 1;
3587          bmp->db_agwidth = 1 << (l2nl - (bmp->db_agheigth << 1));
3588          for (i = 5 - bmp->db_agheigth, bmp->db_agstart = 0, n = 1; i > 0;
3589               i--) {
3590                  bmp->db_agstart += n;
3591                  n <<= 2;
3592          }
3593
3594  /*
3595  printk("bmap: agpref:%d aglevel:%d agheigth:%d agwidth:%d\n",
3596          bmp->db_agpref, bmp->db_aglevel, bmp->db_agheigth, bmp->db_agwidth);
3597   */
3598  }
3599
3600
```

```
3601 /*
3602  * NAME:        dbInitDmap()/ujfs_idmap_page()
3603  *
3604  * FUNCTION:    initialize working/persistent bitmap of the dmap page
3605  *              for the specified number of blocks:
3606  *
3607  *              at entry, the bitmaps had been initialized as free (ZEROS);
3608  *              The number of blocks will only account for the actually
3609  *              existing blocks. Blocks which don't actually exist in
3610  *              the aggregate will be marked as allocated (ONES);
3611  *
3612  * PARAMETERS:
3613  *      dp      − pointer to page of map
3614  *      nblocks − number of blocks this page
3615  *
3616  * RETURNS: NONE
3617  */
3618 static int dbInitDmap(struct dmap * dp, s64 Blkno, int nblocks)
3619 {
3620         int blkno, w, b, r, nw, nb, i;
3621 /*
3622 printk("sbh_dmap:  in dbInitDmap blkno:%Ld nblocks:%ld\n", Blkno, nblocks);
3623 */
3624
3625         /* starting block number within the dmap */
3626         blkno = Blkno & (BPERDMAP − 1);
3627
3628         if (blkno == 0) {
3629                 dp−>nblocks = dp−>nfree = cpu_to_le32(nblocks);
3630                 dp−>start = cpu_to_le64(Blkno);
3631
3632                 if (nblocks == BPERDMAP) {
3633                         memset(&dp−>wmap[0], 0, LPERDMAP * 4);
3634                         memset(&dp−>pmap[0], 0, LPERDMAP * 4);
3635                         goto initTree;
3636                 }
3637         } else {
3638                 dp−>nblocks =
3639                     cpu_to_le32(le32_to_cpu(dp−>nblocks) + nblocks);
3640                 dp−>nfree = cpu_to_le32(le32_to_cpu(dp−>nfree) + nblocks);
3641         }
3642
3643         /* word number containing start block number */
3644         w = blkno >> L2DBWORD;
3645
3646         /*
3647          * free the bits corresponding to the block range (ZEROS):
3648          * note: not all bits of the first and last words may be contained
3649          * within the block range.
3650          */
3651         for (r = nblocks; r > 0; r −= nb, blkno += nb) {
3652                 /* number of bits preceding range to be freed in the word */
3653                 b = blkno & (DBWORD − 1);
3654                 /* number of bits to free in the word */
3655                 nb = min(r, DBWORD − b);
3656
3657                 /* is partial word to be freed ? */
3658                 if (nb < DBWORD) {
3659                         /* free (set to 0) from the bitmap word */
3660                         dp−>wmap[w] &= cpu_to_le32(~(ONES << (DBWORD − nb)
3661                                                      >> b));
3662                         dp−>pmap[w] &= cpu_to_le32(~(ONES << (DBWORD − nb)
3663                                                      >> b));
3664
3665                         /* skip the word freed */
3666                         w++;
3667                 } else {
3668                         /* free (set to 0) contiguous bitmap words */
3669                         nw = r >> L2DBWORD;
3670                         memset(&dp−>wmap[w], 0, nw * 4);
3671                         memset(&dp−>pmap[w], 0, nw * 4);
3672
3673                         /* skip the words freed */
3674                         nb = nw << L2DBWORD;
3675                         w += nw;
3676                 }
3677         }
3678
3679         /*
3680          * mark bits following the range to be freed (non−existing
3681          * blocks) as allocated (ONES)
3682          */
3683 /*
3684 printk("sbh_dmap:  in dbInitDmap, preparing to mark unbacked, blkno:%ld nblocks:%ld\n",
3685                 blkno, nblocks);
3686 */
3687
3688         if (blkno == BPERDMAP)
3689                 goto initTree;
3690
```

```
3691          /* the first word beyond the end of existing blocks */
3692          w = blkno >> L2DBWORD;
3693
3694          /* does nblocks fall on a 32−bit boundary ? */
3695          b = blkno & (DBWORD − 1);
3696  /*
3697  printk("sbh_dmap:  in dbInitDmap, b:%ld w:%ld mask: %lx\n", b, w, (ONES>>b));
3698  */
3699          if (b) {
3700                  /* mark a partial word allocated */
3701                  dp−>wmap[w] = dp−>pmap[w] = cpu_to_le32(ONES >> b);
3702                  w++;
3703          }
3704
3705          /* set the rest of the words in the page to allocated (ONES) */
3706          for (i = w; i < LPERDMAP; i++)
3707                  dp−>pmap[i] = dp−>wmap[i] = ONES;
3708
3709          /*
3710           * init tree
3711           */
3712        initTree:
3713          return (dbInitDmapTree(dp));
3714  }
3715
3716
3717  /*
3718   * NAME:        dbInitDmapTree()/ujfs_complete_dmap()
3719   *
3720   * FUNCTION:    initialize summary tree of the specified dmap:
3721   *
3722   *              at entry, bitmap of the dmap has been initialized;
3723   *
3724   * PARAMETERS:
3725   *      dp      − dmap to complete
3726   *      blkno   − starting block number for this dmap
3727   *      treemax − will be filled in with max free for this dmap
3728   *
3729   * RETURNS:     max free string at the root of the tree
3730   */
3731  static int dbInitDmapTree(struct dmap * dp)
3732  {
3733          struct dmaptree *tp;
3734          s8 *cp;
3735          int i;
3736
3737          /* init fixed info of tree */
3738          tp = &dp−>tree;
3739          tp−>nleafs = cpu_to_le32(LPERDMAP);
3740          tp−>l2nleafs = cpu_to_le32(L2LPERDMAP);
3741          tp−>leafidx = cpu_to_le32(LEAFIND);
3742          tp−>height = cpu_to_le32(4);
3743          tp−>budmin = BUDMIN;
3744
3745          /* init each leaf from corresponding wmap word:
3746           * note: leaf is set to NOFREE(−1) if all blocks of corresponding
3747           * bitmap word are allocated.
3748           */
3749          cp = tp−>stree + le32_to_cpu(tp−>leafidx);
3750          for (i = 0; i < LPERDMAP; i++)
3751                  *cp++ = dbMaxBud((u8 *) & dp−>wmap[i]);
3752
3753          /* build the dmap's binary buddy summary tree */
3754          return (dbInitTree(tp));
3755  }
3756
3757
3758  /*
3759   * NAME:        dbInitTree()/ujfs_adjtree()
3760   *
3761   * FUNCTION:    initialize binary buddy summary tree of a dmap or dmapctl.
3762   *
3763   *              at entry, the leaves of the tree has been initialized
3764   *              from corresponding bitmap word or root of summary tree
3765   *              of the child control page;
3766   *              configure binary buddy system at the leaf level, then
3767   *              bubble up the values of the leaf nodes up the tree.
3768   *
3769   * PARAMETERS:
3770   *      cp      − Pointer to the root of the tree
3771   *      l2leaves− Number of leaf nodes as a power of 2
3772   *      l2min   − Number of blocks that can be covered by a leaf
3773   *                as a power of 2
3774   *
3775   * RETURNS: max free string at the root of the tree
3776   */
3777  static int dbInitTree(struct dmaptree * dtp)
3778  {
3779          int l2max, l2free, bsize, nextb, i;
3780          int child, parent, nparent;
```

```
3781          s8 *tp, *cp, *cp1;
3782
3783          tp = dtp->stree;
3784
3785          /* Determine the maximum free string possible for the leaves */
3786          l2max = le32_to_cpu(dtp->l2nleafs) + dtp->budmin;
3787
3788          /*
3789           * configure the leaf levevl into binary buddy system
3790           *
3791           * Try to combine buddies starting with a buddy size of 1
3792           * (i.e. two leaves). At a buddy size of 1 two buddy leaves
3793           * can be combined if both buddies have a maximum free of l2min;
3794           * the combination will result in the left-most buddy leaf having
3795           * a maximum free of l2min+1.
3796           * After processing all buddies for a given size, process buddies
3797           * at the next higher buddy size (i.e. current size * 2) and
3798           * the next maximum free (current free + 1).
3799           * This continues until the maximum possible buddy combination
3800           * yields maximum free.
3801           */
3802          for (l2free = dtp->budmin, bsize = 1; l2free < l2max;
3803               l2free++, bsize = nextb) {
3804                 /* get next buddy size == current buddy pair size */
3805                 nextb = bsize << 1;
3806
3807                 /* scan each adjacent buddy pair at current buddy size */
3808                 for (i = 0, cp = tp + le32_to_cpu(dtp->leafidx);
3809                      i < le32_to_cpu(dtp->nleafs);
3810                      i += nextb, cp += nextb) {
3811                         /* coalesce if both adjacent buddies are max free */
3812                         if (*cp == l2free && *(cp + bsize) == l2free) {
3813                                 *cp = l2free + 1;       /* left take right */
3814                                 *(cp + bsize) = -1;     /* right give left */
3815                         }
3816                 }
3817          }
3818
3819          /*
3820           * bubble summary information of leaves up the tree.
3821           *
3822           * Starting at the leaf node level, the four nodes described by
3823           * the higher level parent node are compared for a maximum free and
3824           * this maximum becomes the value of the parent node.
3825           * when all lower level nodes are processed in this fashion then
3826           * move up to the next level (parent becomes a lower level node) and
3827           * continue the process for that level.
3828           */
3829          for (child = le32_to_cpu(dtp->leafidx),
3830               nparent = le32_to_cpu(dtp->nleafs) >> 2;
3831               nparent > 0; nparent >>= 2, child = parent) {
3832                 /* get index of 1st node of parent level */
3833                 parent = (child - 1) >> 2;
3834
3835                 /* set the value of the parent node as the maximum
3836                  * of the four nodes of the current level.
3837                  */
3838                 for (i = 0, cp = tp + child, cp1 = tp + parent;
3839                      i < nparent; i++, cp += 4, cp1++)
3840                         *cp1 = TREEMAX(cp);
3841          }
3842
3843          return (*tp);
3844  }
3845
3846
3847  /*
3848   *      dbInitDmapCtl()
3849   *
3850   * function: initialize dmapctl page
3851   */
3852  static int dbInitDmapCtl(struct dmapctl * dcp, int level, int i)
3853  {                               /* start leaf index not covered by range */
3854          s8 *cp;
3855
3856          dcp->nleafs = cpu_to_le32(LPERCTL);
3857          dcp->l2nleafs = cpu_to_le32(L2LPERCTL);
3858          dcp->leafidx = cpu_to_le32(CTLLEAFIND);
3859          dcp->height = cpu_to_le32(5);
3860          dcp->budmin = L2BPERDMAP + L2LPERCTL * level;
3861
3862          /*
3863           * initialize the leaves of current level that were not covered
3864           * by the specified input block range (i.e. the leaves have no
3865           * low level dmapctl or dmap).
3866           */
3867          cp = &dcp->stree[CTLLEAFIND + i];
3868          for (; i < LPERCTL; i++)
3869                  *cp++ = NOFREE;
3870
```

```
3871          /* build the dmap's binary buddy summary tree */
3872          return (dbInitTree((struct dmaptree *) dcp));
3873  }
3874
3875
3876  /*
3877   * NAME:        dbGetL2AGSize()/ujfs_getagl2size()
3878   *
3879   * FUNCTION:    Determine log2(allocation group size) from aggregate size
3880   *
3881   * PARAMETERS:
3882   *      nblocks − Number of blocks in aggregate
3883   *
3884   * RETURNS: log2(allocation group size) in aggregate blocks
3885   */
3886  static int dbGetL2AGSize(s64 nblocks)
3887  {
3888          s64 sz;
3889          s64 m;
3890          int l2sz;
3891
3892          if (nblocks < BPERDMAP * MAXAG)
3893                  return (L2BPERDMAP);
3894
3895          /* round up aggregate size to power of 2 */
3896          m = ((u64) 1 << (64 − 1));
3897          for (l2sz = 64; l2sz >= 0; l2sz−−, m >>= 1) {
3898                  if (m & nblocks)
3899                          break;
3900          }
3901
3902          sz = (s64) 1 << l2sz;
3903          if (sz < nblocks)
3904                  l2sz += 1;
3905
3906          /* agsize = roundupSize/max_number_of_ag */
3907          return (l2sz − L2MAXAG);
3908  }
3909
3910
3911  /*
3912   * NAME:        dbMapFileSizeToMapSize()
3913   *
3914   * FUNCTION:    compute number of blocks the block allocation map file
3915   *              can cover from the map file size;
3916   *
3917   * RETURNS:     Number of blocks which can be covered by this block map file;
3918   */
3919
3920  /*
3921   * maximum number of map pages at each level including control pages
3922   */
3923  #define MAXL0PAGES      (1 + LPERCTL)
3924  #define MAXL1PAGES      (1 + LPERCTL * MAXL0PAGES)
3925  #define MAXL2PAGES      (1 + LPERCTL * MAXL1PAGES)
3926
3927  /*
3928   * convert number of map pages to the zero origin top dmapctl level
3929   */
3930  #define BMAPPGTOLEV(npages)     \
3931          (((npages) <= 3 + MAXL0PAGES) ? 0 \
3932         : ((npages) <= 2 + MAXL1PAGES) ? 1 : 2)
3933
3934  s64 dbMapFileSizeToMapSize(struct inode * ipbmap)
3935  {
3936          struct super_block *sb = ipbmap−>i_sb;
3937          s64 nblocks;
3938          s64 npages, ndmaps;
3939          int level, i;
3940          int complete, factor;
3941
3942          nblocks = ipbmap−>i_size >> JFS_SBI(sb)−>l2bsize;
3943          npages = nblocks >> JFS_SBI(sb)−>l2nbperpage;
3944          level = BMAPPGTOLEV(npages);
3945
3946          /* At each level, accumulate the number of dmap pages covered by
3947           * the number of full child levels below it;
3948           * repeat for the last incomplete child level.
3949           */
3950          ndmaps = 0;
3951          npages−−;               /* skip the first global control page */
3952          /* skip higher level control pages above top level covered by map */
3953          npages −= (2 − level);
3954          npages−−;               /* skip top level's control page */
3955          for (i = level; i >= 0; i−−) {
3956                  factor =
3957                      (i == 2) ? MAXL1PAGES : ((i == 1) ? MAXL0PAGES : 1);
3958                  complete = (u32) npages / factor;
3959                  ndmaps += complete * ((i == 2) ? LPERCTL * LPERCTL
3960                                        : ((i == 1) ? LPERCTL : 1));
```

```
3961
3962                    /* pages in last/incomplete child */
3963                    npages = (u32) npages % factor;
3964                    /* skip incomplete child's level control page */
3965                    npages−−;
3966            }
3967
3968            /* convert the number of dmaps into the number of blocks
3969             * which can be covered by the dmaps;
3970             */
3971            nblocks = ndmaps << L2BPERDMAP;
3972
3973            return (nblocks);
3974    }
3975
3976
3977    #ifdef  _JFS_DEBUG_DMAP
3978    /*
3979     *      DBinitmap()
3980     */
3981    static void DBinitmap(s64 size, struct inode *ipbmap, u32 ** results)
3982    {
3983            int npages;
3984            u32 *dbmap, *d;
3985            int n;
3986            s64 lblkno, cur_block;
3987            struct dmap *dp;
3988            struct metapage *mp;
3989
3990            npages = size / 32768;
3991            npages += (size % 32768) ? 1 : 0;
3992
3993            dbmap = (u32 *) xmalloc(npages * 4096, L2PSIZE, kernel_heap);
3994            if (dbmap == NULL)
3995                    assert(0);
3996
3997            for (n = 0, d = dbmap; n < npages; n++, d += 1024)
3998                    bzero(d, 4096);
3999
4000            /* Need to initialize from disk map pages
4001             */
4002            for (d = dbmap, cur_block = 0; cur_block < size;
4003                 cur_block += BPERDMAP, d += LPERDMAP) {
4004                    lblkno = BLKTODMAP(cur_block,
4005                                       JFS_SBI(ipbmap−>i_sb)−>bmap−>
4006                                       db_l2nbperpage);
4007                    mp = read_metapage(ipbmap, lblkno, PSIZE, 0);
4008                    if (mp == NULL) {
4009                            assert(0);
4010                    }
4011                    dp = (struct dmap *) mp−>data;
4012
4013                    for (n = 0; n < LPERDMAP; n++)
4014                            d[n] = le32_to_cpu(dp−>wmap[n]);
4015
4016                    release_metapage(mp);
4017            }
4018
4019            *results = dbmap;
4020    }
4021
4022
4023    /*
4024     *      DBAlloc()
4025     */
4026    void DBAlloc(uint * dbmap, s64 mapsize, s64 blkno, s64 nblocks)
4027    {
4028            int word, nb, bitno;
4029            u32 mask;
4030
4031            assert(blkno > 0 && blkno < mapsize);
4032            assert(nblocks > 0 && nblocks <= mapsize);
4033
4034            assert(blkno + nblocks <= mapsize);
4035
4036            dbmap += (blkno / 32);
4037            while (nblocks > 0) {
4038                    bitno = blkno & (32 − 1);
4039                    nb = min(nblocks, 32 − bitno);
4040
4041                    mask = (0xffffffff << (32 − nb) >> bitno);
4042                    assert((mask & *dbmap) == 0);
4043                    *dbmap |= mask;
4044
4045                    dbmap++;
4046                    blkno += nb;
4047                    nblocks −= nb;
4048            }
4049    }
4050
```

```
4051
4052 /*
4053  *      DBFree()
4054  */
4055 static void DBFree(uint * dbmap, s64 mapsize, s64 blkno, s64 nblocks)
4056 {
4057         int word, nb, bitno;
4058         u32 mask;
4059
4060         assert(blkno > 0 && blkno < mapsize);
4061         assert(nblocks > 0 && nblocks <= mapsize);
4062
4063         assert(blkno + nblocks <= mapsize);
4064
4065         dbmap += (blkno / 32);
4066         while (nblocks > 0) {
4067                 bitno = blkno & (32 − 1);
4068                 nb = min(nblocks, 32 − bitno);
4069
4070                 mask = (0xffffffff << (32 − nb) >> bitno);
4071                 assert((mask & *dbmap) == mask);
4072                 *dbmap &= ~mask;
4073
4074                 dbmap++;
4075                 blkno += nb;
4076                 nblocks −= nb;
4077         }
4078 }
4079
4080
4081 /*
4082  *      DBAllocCK()
4083  */
4084 static void DBAllocCK(uint * dbmap, s64 mapsize, s64 blkno, s64 nblocks)
4085 {
4086         int word, nb, bitno;
4087         u32 mask;
4088
4089         assert(blkno > 0 && blkno < mapsize);
4090         assert(nblocks > 0 && nblocks <= mapsize);
4091
4092         assert(blkno + nblocks <= mapsize);
4093
4094         dbmap += (blkno / 32);
4095         while (nblocks > 0) {
4096                 bitno = blkno & (32 − 1);
4097                 nb = min(nblocks, 32 − bitno);
4098
4099                 mask = (0xffffffff << (32 − nb) >> bitno);
4100                 assert((mask & *dbmap) == mask);
4101
4102                 dbmap++;
4103                 blkno += nb;
4104                 nblocks −= nb;
4105         }
4106 }
4107
4108
4109 /*
4110  *      DBFreeCK()
4111  */
4112 static void DBFreeCK(uint * dbmap, s64 mapsize, s64 blkno, s64 nblocks)
4113 {
4114         int word, nb, bitno;
4115         u32 mask;
4116
4117         assert(blkno > 0 && blkno < mapsize);
4118         assert(nblocks > 0 && nblocks <= mapsize);
4119
4120         assert(blkno + nblocks <= mapsize);
4121
4122         dbmap += (blkno / 32);
4123         while (nblocks > 0) {
4124                 bitno = blkno & (32 − 1);
4125                 nb = min(nblocks, 32 − bitno);
4126
4127                 mask = (0xffffffff << (32 − nb) >> bitno);
4128                 assert((mask & *dbmap) == 0);
4129
4130                 dbmap++;
4131                 blkno += nb;
4132                 nblocks −= nb;
4133         }
4134 }
4135
4136
4137 /*
4138  *      dbPrtMap()
4139  */
4140 static void dbPrtMap(struct bmap * bmp)
```

```
4141 {
4142         printk("  mapsize:  %d%d\n", bmp->db_mapsize);
4143         printk("  nfree:    %d%d\n", bmp->db_nfree);
4144         printk("  numag:    %d\n", bmp->db_numag);
4145         printk("  agsize:   %d%d\n", bmp->db_agsize);
4146         printk("  agl2size: %d\n", bmp->db_agl2size);
4147         printk("  agwidth:  %d\n", bmp->db_agwidth);
4148         printk("  agstart:  %d\n", bmp->db_agstart);
4149         printk("  agheigth: %d\n", bmp->db_agheigth);
4150         printk("  aglevel:  %d\n", bmp->db_aglevel);
4151         printk("  maxlevel: %d\n", bmp->db_maxlevel);
4152         printk("  maxag:    %d\n", bmp->db_maxag);
4153         printk("  agpref:   %d\n", bmp->db_agpref);
4154         printk("  l2nbppg:  %d\n", bmp->db_l2nbperpage);
4155 }
4156
4157
4158 /*
4159  *      dbPrtCtl()
4160  */
4161 static void dbPrtCtl(struct dmapctl * dcp)
4162 {
4163         int i, j, n;
4164
4165         printk("  height:   %08x\n", le32_to_cpu(dcp->height));
4166         printk("  leafidx:  %08x\n", le32_to_cpu(dcp->leafidx));
4167         printk("  budmin:   %08x\n", dcp->budmin);
4168         printk("  nleafs:   %08x\n", le32_to_cpu(dcp->nleafs));
4169         printk("  l2nleafs: %08x\n", le32_to_cpu(dcp->l2nleafs));
4170
4171         printk("\n Tree:\n");
4172         for (i = 0; i < CTLLEAFIND; i += 8) {
4173                 n = min(8, CTLLEAFIND - i);
4174
4175                 for (j = 0; j < n; j++)
4176                         printf(" [%03x]: %02x", i + j,
4177                                (char) dcp->stree[i + j]);
4178                 printf("\n");
4179         }
4180
4181         printk("\n Tree Leaves:\n");
4182         for (i = 0; i < LPERCTL; i += 8) {
4183                 n = min(8, LPERCTL - i);
4184
4185                 for (j = 0; j < n; j++)
4186                         printf(" [%03x]: %02x",
4187                                i + j,
4188                                (char) dcp->stree[i + j + CTLLEAFIND]);
4189                 printf("\n");
4190         }
4191 }
4192 #endif                          /* _JFS_DEBUG_DMAP */
```

```
/*
 *   Copyright (c) International Business Machines Corp., 2000−2002
 *
 *   This program is free software;  you can redistribute it and/or modify
 *   it under the terms of the GNU General Public License as published by
 *   the Free Software Foundation; either version 2 of the License, or
 *   (at your option) any later version.
 *
 *   This program is distributed in the hope that it will be useful,
 *   but WITHOUT ANY WARRANTY;  without even the implied warranty of
 *   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See
 *   the GNU General Public License for more details.
 *
 *   You should have received a copy of the GNU General Public License
 *   along with this program;  if not, write to the Free Software
 *   Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111−1307 USA
 */

#include <linux/fs.h>
#include <linux/slab.h>
#include "jfs_types.h"
#include "jfs_filsys.h"
#include "jfs_unicode.h"
#include "jfs_debug.h"

/*
 * NAME:        jfs_strfromUCS()
 *
 * FUNCTION:    Convert little−endian unicode string to character string
 *
 */
int jfs_strfromUCS_le(char *to, const wchar_t * from,   /* LITTLE ENDIAN */
                      int len, struct nls_table *codepage)
{
        int i;
        int outlen = 0;

        for (i = 0; (i < len) && from[i]; i++) {
                int charlen;
                charlen =
                    codepage->uni2char(le16_to_cpu(from[i]), &to[outlen],
                                       NLS_MAX_CHARSET_SIZE);
                if (charlen > 0) {
                        outlen += charlen;
                } else {
                        to[outlen++] = '?';
                }
        }
        to[outlen] = 0;
        jEVENT(0, ("jfs_strfromUCS returning %d − '%s'\n", outlen, to));
        return outlen;
}

/*
 * NAME:        jfs_strtoUCS()
 *
 * FUNCTION:    Convert character string to unicode string
 *
 */
int jfs_strtoUCS(wchar_t * to,
                 const char *from, int len, struct nls_table *codepage)
{
        int charlen;
        int i;

        jEVENT(0, ("jfs_strtoUCS − '%s'\n", from));

        for (i = 0; len && *from; i++, from += charlen, len −= charlen) {
                charlen = codepage->char2uni(from, len, &to[i]);
                if (charlen < 1) {
                        jERROR(1, ("jfs_strtoUCS: char2uni returned %d.\n",
                                   charlen));
                        jERROR(1, ("charset = %s, char = 0x%x\n",
                                   codepage->charset, (unsigned char) *from));
                        to[i] = 0x003f; /* a question mark */
                        charlen = 1;
                }
        }

        jEVENT(0, (" returning %d\n", i));

        to[i] = 0;
        return i;
}

/*
 * NAME:        get_UCSname()
 *
 * FUNCTION:    Allocate and translate to unicode string
 *
```

```
 */
int get_UCSname(struct component_name * uniName, struct dentry *dentry,
                struct nls_table *nls_tab)
{
        int length = dentry->d_name.len;

        if (length > JFS_NAME_MAX)
                return ENAMETOOLONG;

        uniName->name =
            kmalloc((length + 1) * sizeof(wchar_t), GFP_NOFS);

        if (uniName->name == NULL)
                return ENOSPC;

        uniName->namlen = jfs_strtoUCS(uniName->name, dentry->d_name.name,
                                       length, nls_tab);

        return 0;
}
```

Table of Content