File: ffs_alloc.c

```c
/*	$OpenBSD: ffs_alloc.c,v 1.113 2020/06/20 07:49:04 otto Exp $	*/
/*	$NetBSD: ffs_alloc.c,v 1.11 1996/05/11 18:27:09 mycroft Exp $	*/

/*
 * Copyright (c) 2002 Networks Associates Technology, Inc.
 * All rights reserved.
 *
 * This software was developed for the FreeBSD Project by Marshall
 * Kirk McKusick and Network Associates Laboratories, the Security
 * Research Division of Network Associates, Inc. under DARPA/SPAWAR
 * contract N66001-01-C-8035 ("CBOSS"), as part of the DARPA CHATS
 * research program.
 *
 * Copyright (c) 1982, 1986, 1989, 1993
 *	The Regents of the University of California.  All rights reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions
 * are met:
 * 1. Redistributions of source code must retain the above copyright
 *    notice, this list of conditions and the following disclaimer.
 * 2. Redistributions in binary form must reproduce the above copyright
 *    notice, this list of conditions and the following disclaimer in the
 *    documentation and/or other materials provided with the distribution.
 * 3. Neither the name of the University nor the names of its contributors
 *    may be used to endorse or promote products derived from this software
 *    without specific prior written permission.
 *
 * THIS SOFTWARE IS PROVIDED BY THE REGENTS AND CONTRIBUTORS ``AS IS'' AND
 * ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
 * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
 * ARE DISCLAIMED.  IN NO EVENT SHALL THE REGENTS OR CONTRIBUTORS BE LIABLE
 * FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
 * DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
 * OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
 * HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
 * LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
 * OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
 * SUCH DAMAGE.
 *
 *	@(#)ffs_alloc.c	8.11 (Berkeley) 10/27/94
 */

#include <sys/param.h>
#include <sys/systm.h>
#include <sys/buf.h>
#include <sys/vnode.h>
#include <sys/mount.h>
#include <sys/syslog.h>
#include <sys/stdint.h>
#include <sys/time.h>

#include <ufs/ufs/quota.h>
#include <ufs/ufs/inode.h>
#include <ufs/ufs/ufsmount.h>
#include <ufs/ufs/ufs_extern.h>

#include <ufs/ffs/fs.h>
#include <ufs/ffs/ffs_extern.h>

#define ffs_fserr(fs, uid, cp) do {				\
	log(LOG_ERR, "uid %u on %s: %s\n", (uid),		\
	    (fs)->fs_fsmnt, (cp));				\
} while (0)

daddr_t		ffs_alloccg(struct inode *, u_int, daddr_t, int);
struct buf *	ffs_cgread(struct fs *, struct inode *, u_int);
daddr_t		ffs_alloccgblk(struct inode *, struct buf *, daddr_t);
ufsino_t	ffs_dirpref(struct inode *);
daddr_t		ffs_fragextend(struct inode *, u_int, daddr_t, int, int);
daddr_t		ffs_hashalloc(struct inode *, u_int, daddr_t, int,
		    daddr_t (*)(struct inode *, u_int, daddr_t, int));
daddr_t		ffs_nodealloccg(struct inode *, u_int, daddr_t, int);
daddr_t		ffs_mapsearch(struct fs *, struct cg *, daddr_t, int);

static const struct timeval	fserr_interval = { 2, 0 };


/*
 * Allocate a block in the file system.
 *
 * The size of the requested block is given, which must be some
 * multiple of fs_fsize and <= fs_bsize.
 * A preference may be optionally specified. If a preference is given
 * the following hierarchy is used to allocate a block:
 *   1) allocate the requested block.
 *   2) allocate a rotationally optimal block in the same cylinder.
 *   3) allocate a block in the same cylinder group.
 *   4) quadratically rehash into other cylinder groups, until an
 *      available block is located.
 * If no block preference is given the following hierarchy is used
 * to allocate a block:
 *   1) allocate a block in the cylinder group that contains the
 *      inode for the file.
 *   2) quadratically rehash into other cylinder groups, until an
 *      available block is located.
 */
int
ffs_alloc(struct inode *ip, daddr_t lbn, daddr_t bpref, int size,
    struct ucred *cred, daddr_t *bnp)
{
	static struct timeval fsfull_last;
	struct fs *fs;
	daddr_t bno;
	u_int cg;
	int error;

	*bnp = 0;
	fs = ip->i_fs;
#ifdef DIAGNOSTIC
	if ((u_int)size > fs->fs_bsize || fragoff(fs, size) != 0) {
		printf("dev = 0x%x, bsize = %d, size = %d, fs = %s\n",
		    ip->i_dev, fs->fs_bsize, size, fs->fs_fsmnt);
		panic("ffs_alloc: bad size");
	}
	if (cred == NOCRED)
		panic("ffs_alloc: missing credential");
#endif /* DIAGNOSTIC */
	if (size == fs->fs_bsize && fs->fs_cstotal.cs_nbfree == 0)
		goto nospace;
	if (cred->cr_uid != 0 && freespace(fs, fs->fs_minfree) <= 0)
		goto nospace;

	if ((error = ufs_quota_alloc_blocks(ip, btodb(size), cred)) != 0)
		return (error);

	/*
	 * Start allocation in the preferred block's cylinder group or
	 * the file's inode's cylinder group if no preferred block was
	 * specified.
	 */
	if (bpref >= fs->fs_size)
		bpref = 0;
	if (bpref == 0)
		cg = ino_to_cg(fs, ip->i_number);
	else
		cg = dtog(fs, bpref);

	/* Try allocating a block. */
	bno = ffs_hashalloc(ip, cg, bpref, size, ffs_alloccg);
	if (bno > 0) {
		/* allocation successful, update inode data */
		DIP_ADD(ip, blocks, btodb(size));
		ip->i_flag |= IN_CHANGE | IN_UPDATE;
		*bnp = bno;
		return (0);
	}

	/* Restore user's disk quota because allocation failed. */
	(void) ufs_quota_free_blocks(ip, btodb(size), cred);

nospace:
	if (ratecheck(&fsfull_last, &fserr_interval)) {
		ffs_fserr(fs, cred->cr_uid, "file system full");
		uprintf("\n%s: write failed, file system is full\n",
		    fs->fs_fsmnt);
	}
	return (ENOSPC);
}

/*
 * Reallocate a fragment to a bigger size
 *
 * The number and size of the old block is given, and a preference
 * and new size is also specified. The allocator attempts to extend
 * the original block. Failing that, the regular block allocator is
 * invoked to get an appropriate block.
 */
int
ffs_realloccg(struct inode *ip, daddr_t lbprev, daddr_t bpref, int osize,
    int nsize, struct ucred *cred, struct buf **bpp, daddr_t *blknop)
{
	static struct timeval fsfull_last;
	struct fs *fs;
	struct buf *bp = NULL;
	daddr_t quota_updated = 0;
	int request, error;
	u_int cg;
	daddr_t bprev, bno;

	if (bpp != NULL)
		*bpp = NULL;
	fs = ip->i_fs;
#ifdef DIAGNOSTIC
	if ((u_int)osize > fs->fs_bsize || fragoff(fs, osize) != 0 ||
	    (u_int)nsize > fs->fs_bsize || fragoff(fs, nsize) != 0) {
		printf(
		    "dev = 0x%x, bsize = %d, osize = %d, nsize = %d, fs = %s\n",
		    ip->i_dev, fs->fs_bsize, osize, nsize, fs->fs_fsmnt);
		panic("ffs_realloccg: bad size");
	}
	if (cred == NOCRED)
		panic("ffs_realloccg: missing credential");
#endif /* DIAGNOSTIC */
	if (cred->cr_uid != 0 && freespace(fs, fs->fs_minfree) <= 0)
		goto nospace;

	bprev = DIP(ip, db[lbprev]);

	if (bprev == 0) {
		printf("dev = 0x%x, bsize = %d, bprev = %lld, fs = %s\n",
		    ip->i_dev, fs->fs_bsize, (long long)bprev, fs->fs_fsmnt);
		panic("ffs_realloccg: bad bprev");
	}

	/*
	 * Allocate the extra space in the buffer.
	 */
	if (bpp != NULL) {
		if ((error = bread(ITOV(ip), lbprev, fs->fs_bsize, &bp)) != 0)
			goto error;
		buf_adjcnt(bp, osize);
	}

	if ((error = ufs_quota_alloc_blocks(ip, btodb(nsize - osize), cred))
	    != 0)
		goto error;

	quota_updated = btodb(nsize - osize);

	/*
	 * Check for extension in the existing location.
	 */
	cg = dtog(fs, bprev);
	if ((bno = ffs_fragextend(ip, cg, bprev, osize, nsize)) != 0) {
		DIP_ADD(ip, blocks, btodb(nsize - osize));
		ip->i_flag |= IN_CHANGE | IN_UPDATE;
		if (bpp != NULL) {
			if (bp->b_blkno != fsbtodb(fs, bno))
				panic("ffs_realloccg: bad blockno");
#ifdef DIAGNOSTIC
			if (nsize > bp->b_bufsize)
				panic("ffs_realloccg: small buf");
#endif
			buf_adjcnt(bp, nsize);
			bp->b_flags |= B_DONE;
			memset(bp->b_data + osize, 0, nsize - osize);
			*bpp = bp;
		}
		if (blknop != NULL) {
			*blknop = bno;
		}
		return (0);
	}
	/*
	 * Allocate a new disk location.
	 */
	if (bpref >= fs->fs_size)
		bpref = 0;
	switch (fs->fs_optim) {
	case FS_OPTSPACE:
		/*
		 * Allocate an exact sized fragment. Although this makes
		 * best use of space, we will waste time relocating it if
		 * the file continues to grow. If the fragmentation is
		 * less than half of the minimum free reserve, we choose
		 * to begin optimizing for time.
		 */
		request = nsize;
		if (fs->fs_minfree < 5 ||
		    fs->fs_cstotal.cs_nffree >
		    fs->fs_dsize * fs->fs_minfree / (2 * 100))
			break;
		fs->fs_optim = FS_OPTTIME;
		break;
	case FS_OPTTIME:
		/*
		 * At this point we have discovered a file that is trying to
		 * grow a small fragment to a larger fragment. To save time,
		 * we allocate a full sized block, then free the unused portion.
		 * If the file continues to grow, the `ffs_fragextend' call
		 * above will be able to grow it in place without further
		 * copying. If aberrant programs cause disk fragmentation to
		 * grow within 2% of the free reserve, we choose to begin
		 * optimizing for space.
		 */
		request = fs->fs_bsize;
		if (fs->fs_cstotal.cs_nffree <
		    fs->fs_dsize * (fs->fs_minfree - 2) / 100)
			break;
		fs->fs_optim = FS_OPTSPACE;
		break;
	default:
		printf("dev = 0x%x, optim = %d, fs = %s\n",
		    ip->i_dev, fs->fs_optim, fs->fs_fsmnt);
		panic("ffs_realloccg: bad optim");
		/* NOTREACHED */
	}
	bno = ffs_hashalloc(ip, cg, bpref, request, ffs_alloccg);
	if (bno <= 0)
		goto nospace;

	(void) uvm_vnp_uncache(ITOV(ip));
	if (!DOINGSOFTDEP(ITOV(ip)))
		ffs_blkfree(ip, bprev, (long)osize);
	if (nsize < request)
		ffs_blkfree(ip, bno + numfrags(fs, nsize),
		    (long)(request - nsize));
	DIP_ADD(ip, blocks, btodb(nsize - osize));
	ip->i_flag |= IN_CHANGE | IN_UPDATE;
	if (bpp != NULL) {
		bp->b_blkno = fsbtodb(fs, bno);
#ifdef DIAGNOSTIC
		if (nsize > bp->b_bufsize)
			panic("ffs_realloccg: small buf 2");
#endif
		buf_adjcnt(bp, nsize);
		bp->b_flags |= B_DONE;
		memset(bp->b_data + osize, 0, nsize - osize);
		*bpp = bp;
	}
	if (blknop != NULL) {
		*blknop = bno;
	}
	return (0);

nospace:
	if (ratecheck(&fsfull_last, &fserr_interval)) {
		ffs_fserr(fs, cred->cr_uid, "file system full");
		uprintf("\n%s: write failed, file system is full\n",
		    fs->fs_fsmnt);
	}
	error = ENOSPC;

error:
	if (bp != NULL) {
		brelse(bp);
		bp = NULL;
	}

	/*
	 * Restore user's disk quota because allocation failed.
	 */
	if (quota_updated != 0)
		(void)ufs_quota_free_blocks(ip, quota_updated, cred);

	return error;
}

/*
 * Allocate an inode in the file system.
 *
 * If allocating a directory, use ffs_dirpref to select the inode.
 * If allocating in a directory, the following hierarchy is followed:
 *   1) allocate the preferred inode.
 *   2) allocate an inode in the same cylinder group.
 *   3) quadratically rehash into other cylinder groups, until an
 *      available inode is located.
 * If no inode preference is given the following hierarchy is used
 * to allocate an inode:
 *   1) allocate an inode in cylinder group 0.
 *   2) quadratically rehash into other cylinder groups, until an
 *      available inode is located.
 */
int
ffs_inode_alloc(struct inode *pip, mode_t mode, struct ucred *cred,
    struct vnode **vpp)
{
	static struct timeval fsnoinodes_last;
	struct vnode *pvp = ITOV(pip);
	struct fs *fs;
	struct inode *ip;
	ufsino_t ino, ipref;
	u_int cg;
	int error;

	*vpp = NULL;
	fs = pip->i_fs;
	if (fs->fs_cstotal.cs_nifree == 0)
		goto noinodes;

	if ((mode & IFMT) == IFDIR)
		ipref = ffs_dirpref(pip);
	else
		ipref = pip->i_number;
	if (ipref >= fs->fs_ncg * fs->fs_ipg)
		ipref = 0;
	cg = ino_to_cg(fs, ipref);

	/*
	 * Track number of dirs created one after another
	 * in a same cg without intervening by files.
	 */
	if ((mode & IFMT) == IFDIR) {
		if (fs->fs_contigdirs[cg] < 255)
			fs->fs_contigdirs[cg]++;
	} else {
		if (fs->fs_contigdirs[cg] > 0)
			fs->fs_contigdirs[cg]--;
	}
	ino = (ufsino_t)ffs_hashalloc(pip, cg, ipref, mode, ffs_nodealloccg);
	if (ino == 0)
		goto noinodes;
	error = VFS_VGET(pvp->v_mount, ino, vpp);
	if (error) {
		ffs_inode_free(pip, ino, mode);
		return (error);
	}

	ip = VTOI(*vpp);

	if (DIP(ip, mode)) {
		printf("mode = 0%o, inum = %u, fs = %s\n",
		    DIP(ip, mode), ip->i_number, fs->fs_fsmnt);
		panic("ffs_valloc: dup alloc");
	}

	if (DIP(ip, blocks)) {
		printf("free inode %s/%d had %lld blocks\n",
		    fs->fs_fsmnt, ino, (long long)DIP(ip, blocks));
		DIP_ASSIGN(ip, blocks, 0);
	}

	DIP_ASSIGN(ip, flags, 0);

	/*
	 * Set up a new generation number for this inode.
	 * On wrap, we make sure to assign a number != 0 and != UINT_MAX
	 * (the origial value).
	 */
	if (DIP(ip, gen) != 0)
		DIP_ADD(ip, gen, 1);
	while (DIP(ip, gen) == 0)
		DIP_ASSIGN(ip, gen, arc4random_uniform(UINT_MAX));

	return (0);

noinodes:
	if (ratecheck(&fsnoinodes_last, &fserr_interval)) {
		ffs_fserr(fs, cred->cr_uid, "out of inodes");
		uprintf("\n%s: create/symlink failed, no inodes free\n",
		    fs->fs_fsmnt);
	}
	return (ENOSPC);
}

/*
 * Find a cylinder group to place a directory.
 *
 * The policy implemented by this algorithm is to allocate a
 * directory inode in the same cylinder group as its parent
 * directory, but also to reserve space for its files inodes
 * and data. Restrict the number of directories which may be
 * allocated one after another in the same cylinder group
 * without intervening allocation of files.
 *
 * If we allocate a first level directory then force allocation
 * in another cylinder group.
 */
ufsino_t
ffs_dirpref(struct inode *pip)
{
	struct fs *fs;
	u_int	cg, prefcg;
	u_int	dirsize, cgsize;
	u_int	avgifree, avgbfree, avgndir, curdirsize;
	u_int	minifree, minbfree, maxndir;
	u_int	mincg, minndir;
	u_int	maxcontigdirs;

	fs = pip->i_fs;

	avgifree = fs->fs_cstotal.cs_nifree / fs->fs_ncg;
	avgbfree = fs->fs_cstotal.cs_nbfree / fs->fs_ncg;
	avgndir = fs->fs_cstotal.cs_ndir / fs->fs_ncg;

	/*
	 * Force allocation in another cg if creating a first level dir.
	 */
	if (ITOV(pip)->v_flag & VROOT) {
		prefcg = arc4random_uniform(fs->fs_ncg);
		mincg = prefcg;
		minndir = fs->fs_ipg;
		for (cg = prefcg; cg < fs->fs_ncg; cg++)
			if (fs->fs_cs(fs, cg).cs_ndir < minndir &&
			    fs->fs_cs(fs, cg).cs_nifree >= avgifree &&
			    fs->fs_cs(fs, cg).cs_nbfree >= avgbfree) {
				mincg = cg;
				minndir = fs->fs_cs(fs, cg).cs_ndir;
			}
		for (cg = 0; cg < prefcg; cg++)
			if (fs->fs_cs(fs, cg).cs_ndir < minndir &&
			    fs->fs_cs(fs, cg).cs_nifree >= avgifree &&
			    fs->fs_cs(fs, cg).cs_nbfree >= avgbfree) {
				mincg = cg;
				minndir = fs->fs_cs(fs, cg).cs_ndir;
			}
		cg = mincg;
		goto end;
	} else
		prefcg = ino_to_cg(fs, pip->i_number);

	/*
	 * Count various limits which used for
	 * optimal allocation of a directory inode.
	 */
	maxndir = min(avgndir + fs->fs_ipg / 16, fs->fs_ipg);
	minifree = avgifree - (avgifree / 4);
	if (minifree < 1)
		minifree = 1;
	minbfree = avgbfree - (avgbfree / 4);
	if (minbfree < 1)
		minbfree = 1;

	cgsize = fs->fs_fsize * fs->fs_fpg;
	dirsize = fs->fs_avgfilesize * fs->fs_avgfpdir;
	curdirsize = avgndir ? (cgsize - avgbfree * fs->fs_bsize) / avgndir : 0;
	if (dirsize < curdirsize)
		dirsize = curdirsize;
	if (dirsize <= 0)
		maxcontigdirs = 0;		/* dirsize overflowed */
	else
		maxcontigdirs = min(avgbfree * fs->fs_bsize  / dirsize, 255);
	if (fs->fs_avgfpdir > 0)
		maxcontigdirs = min(maxcontigdirs,
				    fs->fs_ipg / fs->fs_avgfpdir);
	if (maxcontigdirs == 0)
		maxcontigdirs = 1;

	/*
	 * Limit number of dirs in one cg and reserve space for
	 * regular files, but only if we have no deficit in
	 * inodes or space.
	 *
	 * We are trying to find a suitable cylinder group nearby
	 * our preferred cylinder group to place a new directory.
	 * We scan from our preferred cylinder group forward looking
	 * for a cylinder group that meets our criterion. If we get
	 * to the final cylinder group and do not find anything,
	 * we start scanning forwards from the beginning of the
	 * filesystem. While it might seem sensible to start scanning
	 * backwards or even to alternate looking forward and backward,
	 * this approach fails badly when the filesystem is nearly full.
	 * Specifically, we first search all the areas that have no space
	 * and finally try the one preceding that. We repeat this on
	 * every request and in the case of the final block end up
	 * searching the entire filesystem. By jumping to the front
	 * of the filesystem, our future forward searches always look
	 * in new cylinder groups so finds every possible block after
	 * one pass over the filesystem.
	 */
	for (cg = prefcg; cg < fs->fs_ncg; cg++)
		if (fs->fs_cs(fs, cg).cs_ndir < maxndir &&
		    fs->fs_cs(fs, cg).cs_nifree >= minifree &&
		    fs->fs_cs(fs, cg).cs_nbfree >= minbfree) {
			if (fs->fs_contigdirs[cg] < maxcontigdirs)
				goto end;
		}
	for (cg = 0; cg < prefcg; cg++)
		if (fs->fs_cs(fs, cg).cs_ndir < maxndir &&
		    fs->fs_cs(fs, cg).cs_nifree >= minifree &&
		    fs->fs_cs(fs, cg).cs_nbfree >= minbfree) {
			if (fs->fs_contigdirs[cg] < maxcontigdirs)
				goto end;
		}
	/*
	 * This is a backstop when we have deficit in space.
	 */
	for (cg = prefcg; cg < fs->fs_ncg; cg++)
		if (fs->fs_cs(fs, cg).cs_nifree >= avgifree)
			goto end;
	for (cg = 0; cg < prefcg; cg++)
		if (fs->fs_cs(fs, cg).cs_nifree >= avgifree)
			goto end;
end:
	return ((ufsino_t)(fs->fs_ipg * cg));
}

/*
 * Select the desired position for the next block in a file.  The file is
 * logically divided into sections. The first section is composed of the
 * direct blocks. Each additional section contains fs_maxbpg blocks.
 *
 * If no blocks have been allocated in the first section, the policy is to
 * request a block in the same cylinder group as the inode that describes
 * the file. The first indirect is allocated immediately following the last
 * direct block and the data blocks for the first indirect immediately
 * follow it.
 *
 * If no blocks have been allocated in any other section, the indirect
 * block(s) are allocated in the same cylinder group as its inode in an
 * area reserved immediately following the inode blocks. The policy for
 * the data blocks is to place them in a cylinder group with a greater than
 * average number of free blocks. An appropriate cylinder group is found
 * by using a rotor that sweeps the cylinder groups. When a new group of
 * blocks is needed, the sweep begins in the cylinder group following the
 * cylinder group from which the previous allocation was made. The sweep
 * continues until a cylinder group with greater than the average number
 * of free blocks is found. If the allocation is for the first block in an
 * indirect block, the information on the previous allocation is unavailable;
 * here a best guess is made based upon the logical block number being
 * allocated.
 */
int32_t
ffs1_blkpref(struct inode *ip, daddr_t lbn, int indx, int32_t *bap)
{
	struct fs *fs;
	u_int cg, inocg;
	u_int avgbfree, startcg;
	uint32_t pref;

	KASSERT(indx <= 0 || bap != NULL);
	fs = ip->i_fs;
	/*
	 * Allocation of indirect blocks is indicated by passing negative
```

```c
         * values in indx: -1 for single indirect, -2 for double indirect,
         * -3 for triple indirect. As noted below, we attempt to allocate
         * the first indirect inline with the file data. For all later
         * indirect blocks, the data is often allocated in other cylinder
         * groups. However to speed random file access and to speed up
         * fsck, the filesystem reserves the first fs_metaspace blocks
         * (typically half of fs_minfree) of the data area of each cylinder
         * group to hold these later indirect blocks.
         */
        inocg = ino_to_cg(fs, ip->i_number);
        if (indx < 0) {
                /*
                 * Our preference for indirect blocks is the zone at the
                 * beginning of the inode's cylinder group data area that
                 * we try to reserve for indirect blocks.
                 */
                pref = cgmeta(fs, inocg);
                /*
                 * If we are allocating the first indirect block, try to
                 * place it immediately following the last direct block.
                 */
                if (indx == -1 && lbn < NDADDR + NINDIR(fs) &&
                    ip->i_din1->di_db[NDADDR - 1] != 0)
                        pref = ip->i_din1->di_db[NDADDR - 1] + fs->fs_frag;
                return (pref);
        }
        /*
         * If we are allocating the first data block in the first indirect
         * block and the indirect has been allocated in the data block area,
         * try to place it immediately following the indirect block.
         */
        if (lbn == NDADDR) {
                pref = ip->i_din1->di_ib[0];
                if (pref != 0 && pref >= cgdata(fs, inocg) &&
                    pref < cgbase(fs, inocg + 1))
                        return (pref + fs->fs_frag);
        }
        /*
         * If we are the beginning of a file, or we have already allocated
         * the maximum number of blocks per cylinder group, or we do not
         * have a block allocated immediately preceding us, then we need
         * to decide where to start allocating new blocks.
         */
        if (indx % fs->fs_maxbpg == 0 || bap[indx - 1] == 0) {
                /*
                 * If we are allocating a directory data block, we want
                 * to place it in the metadata area.
                 */
                if ((DIP(ip, mode) & IFMT) == IFDIR)
                        return (cgmeta(fs, inocg));
                /*
                 * Until we fill all the direct and all the first indirect's
                 * blocks, we try to allocate in the data area of the inode's
                 * cylinder group.
                 */
                if (lbn < NDADDR + NINDIR(fs))
                        return (cgdata(fs, inocg));
                /*
                 * Find a cylinder with greater than average number of
                 * unused data blocks.
                 */
                if (indx == 0 || bap[indx - 1] == 0)
                        startcg = inocg + lbn / fs->fs_maxbpg;
                else
                        startcg = dtog(fs, bap[indx - 1]) + 1;
                startcg %= fs->fs_ncg;
                avgbfree = fs->fs_cstotal.cs_nbfree / fs->fs_ncg;
                for (cg = startcg; cg < fs->fs_ncg; cg++)
                        if (fs->fs_cs(fs, cg).cs_nbfree >= avgbfree) {
                                fs->fs_cgrotor = cg;
                                return (cgdata(fs, cg));
                        }
                for (cg = 0; cg <= startcg; cg++)
                        if (fs->fs_cs(fs, cg).cs_nbfree >= avgbfree) {
                                fs->fs_cgrotor = cg;
                                return (cgdata(fs, cg));
                        }
                return (0);
        }
        /*
         * Otherwise, we just always try to lay things out contiguously.
         */
        return (bap[indx - 1] + fs->fs_frag);
}

/*
 * Same as above, for UFS2.
 */
#ifdef FFS2
int64_t
ffs2_blkpref(struct inode *ip, daddr_t lbn, int indx, int64_t *bap)
{
        struct fs *fs;
        u_int cg, inocg;
        u_int avgbfree, startcg;
        uint64_t pref;

        KASSERT(indx <= 0 || bap != NULL);
        fs = ip->i_fs;
        /*
         * Allocation of indirect blocks is indicated by passing negative
         * values in indx: -1 for single indirect, -2 for double indirect,
         * -3 for triple indirect. As noted below, we attempt to allocate
         * the first indirect inline with the file data. For all later
         * indirect blocks, the data is often allocated in other cylinder
         * groups. However to speed random file access and to speed up
         * fsck, the filesystem reserves the first fs_metaspace blocks
         * (typically half of fs_minfree) of the data area of each cylinder
         * group to hold these later indirect blocks.
         */
        inocg = ino_to_cg(fs, ip->i_number);
        if (indx < 0) {
                /*
                 * Our preference for indirect blocks is the zone at the
                 * beginning of the inode's cylinder group data area that
                 * we try to reserve for indirect blocks.
                 */
                pref = cgmeta(fs, inocg);
                /*
                 * If we are allocating the first indirect block, try to
                 * place it immediately following the last direct block.
                 */
                if (indx == -1 && lbn < NDADDR + NINDIR(fs) &&
                    ip->i_din2->di_db[NDADDR - 1] != 0)
                        pref = ip->i_din2->di_db[NDADDR - 1] + fs->fs_frag;
                return (pref);
        }
        /*
         * If we are allocating the first data block in the first indirect
         * block and the indirect has been allocated in the data block area,
         * try to place it immediately following the indirect block.
         */
        if (lbn == NDADDR) {
                pref = ip->i_din2->di_ib[0];
                if (pref != 0 && pref >= cgdata(fs, inocg) &&
                    pref < cgbase(fs, inocg + 1))
                        return (pref + fs->fs_frag);
        }
        /*
         * If we are the beginning of a file, or we have already allocated
         * the maximum number of blocks per cylinder group, or we do not
         * have a block allocated immediately preceding us, then we need
         * to decide where to start allocating new blocks.
         */

        if (indx % fs->fs_maxbpg == 0 || bap[indx - 1] == 0) {
                /*
                 * If we are allocating a directory data block, we want
                 * to place it in the metadata area.
                 */
                if ((DIP(ip, mode) & IFMT) == IFDIR)
                        return (cgmeta(fs, inocg));
                /*
                 * Until we fill all the direct and all the first indirect's
                 * blocks, we try to allocate in the data area of the inode's
                 * cylinder group.
                 */
                if (lbn < NDADDR + NINDIR(fs))
                        return (cgdata(fs, inocg));
                /*
                 * Find a cylinder with greater than average number of
                 * unused data blocks.
                 */
                if (indx == 0 || bap[indx - 1] == 0)
                        startcg = inocg + lbn / fs->fs_maxbpg;
                else
                        startcg = dtog(fs, bap[indx - 1] + 1);

                startcg %= fs->fs_ncg;
                avgbfree = fs->fs_cstotal.cs_nbfree / fs->fs_ncg;

                for (cg = startcg; cg < fs->fs_ncg; cg++)
                        if (fs->fs_cs(fs, cg).cs_nbfree >= avgbfree)
                                return (cgbase(fs, cg) + fs->fs_frag);

                for (cg = 0; cg < startcg; cg++)
                        if (fs->fs_cs(fs, cg).cs_nbfree >= avgbfree)
                                return (cgbase(fs, cg) + fs->fs_frag);

                return (0);
        }

        /*
         * Otherwise, we just always try to lay things out contiguously.
         */
        return (bap[indx - 1] + fs->fs_frag);
}
#endif /* FFS2 */

/*
 * Implement the cylinder overflow algorithm.
 *
 * The policy implemented by this algorithm is:
 *   1) allocate the block in its requested cylinder group.
 *   2) quadratically rehash on the cylinder group number.
 *   3) brute force search for a free block.
 */
daddr_t
ffs_hashalloc(struct inode *ip, u_int cg, daddr_t pref, int size,
    daddr_t (*allocator)(struct inode *, u_int, daddr_t, int))
{
        struct fs *fs;
        daddr_t result;
        u_int i, icg = cg;

        fs = ip->i_fs;
        /*
         * 1: preferred cylinder group
         */
        result = (*allocator)(ip, cg, pref, size);
        if (result)
                return (result);
        /*
         * 2: quadratic rehash
         */
        for (i = 1; i < fs->fs_ncg; i *= 2) {
                cg += i;
                if (cg >= fs->fs_ncg)
                        cg -= fs->fs_ncg;
                result = (*allocator)(ip, cg, 0, size);
                if (result)
                        return (result);
        }
        /*
         * 3: brute force search
         * Note that we start at i == 2, since 0 was checked initially,
         * and 1 is always checked in the quadratic rehash.
         */
        cg = (icg + 2) % fs->fs_ncg;
        for (i = 2; i < fs->fs_ncg; i++) {
                result = (*allocator)(ip, cg, 0, size);
                if (result)
                        return (result);
                cg++;
                if (cg == fs->fs_ncg)
                        cg = 0;
        }
        return (0);
}

struct buf *
ffs_cgread(struct fs *fs, struct inode *ip, u_int cg)
{
        struct buf *bp;

        if (bread(ip->i_devvp, fsbtodb(fs, cgtod(fs, cg)),
            (int)fs->fs_cgsize, &bp)) {
                brelse(bp);
                return (NULL);
        }

        if (!cg_chkmagic((struct cg *)bp->b_data)) {
                brelse(bp);
                return (NULL);
        }

        return bp;
}

/*
 * Determine whether a fragment can be extended.
 *
 * Check to see if the necessary fragments are available, and
 * if they are, allocate them.
 */
daddr_t
ffs_fragextend(struct inode *ip, u_int cg, daddr_t bprev, int osize, int nsize)
{
        struct fs *fs;
        struct cg *cgp;
        struct buf *bp;
        struct timespec now;
        daddr_t bno;
        int i, frags, bbase;

        fs = ip->i_fs;
        if (fs->fs_cs(fs, cg).cs_nffree < numfrags(fs, nsize - osize))
                return (0);
        frags = numfrags(fs, nsize);
        bbase = fragnum(fs, bprev);
        if (bbase > fragnum(fs, (bprev + frags - 1))) {
                /* cannot extend across a block boundary */
                return (0);
        }

        if (!(bp = ffs_cgread(fs, ip, cg)))
                return (0);

        cgp = (struct cg *)bp->b_data;
        nanotime(&now);
        cgp->cg_ffs2_time = now.tv_sec;
        cgp->cg_time = now.tv_sec;

        bno = dtogd(fs, bprev);
        for (i = numfrags(fs, osize); i < frags; i++)
                if (isclr(cg_blksfree(cgp), bno + i)) {
                        brelse(bp);
                        return (0);
                }
        /*
         * the current fragment can be extended
         * deduct the count on fragment being extended into
         * increase the count on the remaining fragment (if any)
         * allocate the extended piece
```

```
	 */
	for (i = frags; i < fs->fs_frag - bbase; i++)
		if (isclr(cg_blksfree(cgp), bno + i))
			break;
	cgp->cg_frsum[i - numfrags(fs, osize)]--;
	if (i != frags)
		cgp->cg_frsum[i - frags]++;
	for (i = numfrags(fs, osize); i < frags; i++) {
		clrbit(cg_blksfree(cgp), bno + i);
		cgp->cg_cs.cs_nffree--;
		fs->fs_cstotal.cs_nffree--;
		fs->fs_cs(fs, cg).cs_nffree--;
	}
	fs->fs_fmod = 1;
	if (DOINGSOFTDEP(ITOV(ip)))
		softdep_setup_blkmapdep(bp, fs, bprev);

	bdwrite(bp);
	return (bprev);
}

/*
 * Determine whether a block can be allocated.
 *
 * Check to see if a block of the appropriate size is available,
 * and if it is, allocate it.
 */
daddr_t
ffs_alloccg(struct inode *ip, u_int cg, daddr_t bpref, int size)
{
	struct fs *fs;
	struct cg *cgp;
	struct buf *bp;
	struct timespec now;
	daddr_t bno, blkno;
	int i, frags, allocsiz;

	fs = ip->i_fs;
	if (fs->fs_cs(fs, cg).cs_nbfree == 0 && size == fs->fs_bsize)
		return (0);

	if (!(bp = ffs_cgread(fs, ip, cg)))
		return (0);

	cgp = (struct cg *)bp->b_data;
	if (cgp->cg_cs.cs_nbfree == 0 && size == fs->fs_bsize) {
		brelse(bp);
		return (0);
	}

	nanotime(&now);
	cgp->cg_ffs2_time = now.tv_sec;
	cgp->cg_time = now.tv_sec;

	if (size == fs->fs_bsize) {
		/* allocate and return a complete data block */
		bno = ffs_alloccgblk(ip, bp, bpref);
		bdwrite(bp);
		return (bno);
	}
	/*
	 * check to see if any fragments are already available
	 * allocsiz is the size which will be allocated, hacking
	 * it down to a smaller size if necessary
	 */
	frags = numfrags(fs, size);
	for (allocsiz = frags; allocsiz < fs->fs_frag; allocsiz++)
		if (cgp->cg_frsum[allocsiz] != 0)
			break;
	if (allocsiz == fs->fs_frag) {
		/*
		 * no fragments were available, so a block will be
		 * allocated, and hacked up
		 */
		if (cgp->cg_cs.cs_nbfree == 0) {
			brelse(bp);
			return (0);
		}
		bno = ffs_alloccgblk(ip, bp, bpref);
		bpref = dtogd(fs, bno);
		for (i = frags; i < fs->fs_frag; i++)
			setbit(cg_blksfree(cgp), bpref + i);
		i = fs->fs_frag - frags;
		cgp->cg_cs.cs_nffree += i;
		fs->fs_cstotal.cs_nffree += i;
		fs->fs_cs(fs, cg).cs_nffree += i;
		fs->fs_fmod = 1;
		cgp->cg_frsum[i]++;
		bdwrite(bp);
		return (bno);
	}
	bno = ffs_mapsearch(fs, cgp, bpref, allocsiz);
	if (bno < 0) {
		brelse(bp);
		return (0);
	}

	for (i = 0; i < frags; i++)
		clrbit(cg_blksfree(cgp), bno + i);
	cgp->cg_cs.cs_nffree -= frags;
	fs->fs_cstotal.cs_nffree -= frags;
	fs->fs_cs(fs, cg).cs_nffree -= frags;
	fs->fs_fmod = 1;
	cgp->cg_frsum[allocsiz]--;
	if (frags != allocsiz)
		cgp->cg_frsum[allocsiz - frags]++;

	blkno = cgbase(fs, cg) + bno;
	if (DOINGSOFTDEP(ITOV(ip)))
		softdep_setup_blkmapdep(bp, fs, blkno);
	bdwrite(bp);
	return (blkno);
}

/*
 * Allocate a block in a cylinder group.
 * Note that this routine only allocates fs_bsize blocks; these
 * blocks may be fragmented by the routine that allocates them.
 */
daddr_t
ffs_alloccgblk(struct inode *ip, struct buf *bp, daddr_t bpref)
{
	struct fs *fs;
	struct cg *cgp;
	daddr_t bno, blkno;
	u_int8_t *blksfree;
	int cylno, cgbpref;

	fs = ip->i_fs;
	cgp = (struct cg *) bp->b_data;
	blksfree = cg_blksfree(cgp);

	if (bpref == 0) {
		bpref = cgp->cg_rotor;
	} else if ((cgbpref = dtog(fs, bpref)) != cgp->cg_cgx) {
		/* map bpref to correct zone in this cg */
		if (bpref < cgdata(fs, cgbpref))
			bpref = cgmeta(fs, cgp->cg_cgx);
		else
			bpref = cgdata(fs, cgp->cg_cgx);
	}
	/*
	 * If the requested block is available, use it.
	 */
	bno = dtogd(fs, blknum(fs, bpref));
	if (ffs_isblock(fs, blksfree, fragstoblks(fs, bno)))
		goto gotit;
	/*
	 * Take the next available block in this cylinder group.
	 */
	bno = ffs_mapsearch(fs, cgp, bpref, (int) fs->fs_frag);
	if (bno < 0)
		return (0);

	/* Update cg_rotor only if allocated from the data zone */
	if (bno >= dtogd(fs, cgdata(fs, cgp->cg_cgx)))
		cgp->cg_rotor = bno;

gotit:
	blkno = fragstoblks(fs, bno);
	ffs_clrblock(fs, blksfree, blkno);
	ffs_clusteracct(fs, cgp, blkno, -1);
	cgp->cg_cs.cs_nbfree--;
	fs->fs_cstotal.cs_nbfree--;
	fs->fs_cs(fs, cgp->cg_cgx).cs_nbfree--;

	if (fs->fs_magic != FS_UFS2_MAGIC) {
		cylno = cbtocylno(fs, bno);
		cg_blks(fs, cgp, cylno)[cbtorpos(fs, bno)]--;
		cg_blktot(cgp)[cylno]--;
	}

	fs->fs_fmod = 1;
	blkno = cgbase(fs, cgp->cg_cgx) + bno;

	if (DOINGSOFTDEP(ITOV(ip)))
		softdep_setup_blkmapdep(bp, fs, blkno);

	return (blkno);
}

/* inode allocation routine */
daddr_t
ffs_nodealloccg(struct inode *ip, u_int cg, daddr_t ipref, int mode)
{
	struct fs *fs;
	struct cg *cgp;
	struct buf *bp;
	struct timespec now;
	int start, len, loc, map, i;
#ifdef FFS2
	struct buf *ibp = NULL;
	struct ufs2_dinode *dp2;
#endif

	/*
	 * For efficiency, before looking at the bitmaps for free inodes,
	 * check the counters kept in the superblock cylinder group summaries,
	 * and in the cylinder group itself.
	 */
	fs = ip->i_fs;
	if (fs->fs_cs(fs, cg).cs_nifree == 0)
		return (0);

	if (!(bp = ffs_cgread(fs, ip, cg)))
		return (0);

	cgp = (struct cg *)bp->b_data;
	if (cgp->cg_cs.cs_nifree == 0) {
		brelse(bp);
		return (0);
	}

	/*
	 * We are committed to the allocation from now on, so update the time
	 * on the cylinder group.
	 */
	nanotime(&now);
	cgp->cg_ffs2_time = now.tv_sec;
	cgp->cg_time = now.tv_sec;

	/*
	 * If there was a preferred location for the new inode, try to find it.
	 */
	if (ipref) {
		ipref %= fs->fs_ipg;
		if (isclr(cg_inosused(cgp), ipref))
			goto gotit; /* inode is free, grab it. */
	}

	/*
	 * Otherwise, look for the next available inode, starting at cg_irotor
	 * (the position in the bitmap of the last used inode).
	 */
	start = cgp->cg_irotor / NBBY;
	len = howmany(fs->fs_ipg - cgp->cg_irotor, NBBY);
	loc = skpc(0xff, len, &cg_inosused(cgp)[start]);
	if (loc == 0) {
		/*
		 * If we didn't find a free inode in the upper part of the
		 * bitmap (from cg_irotor to the end), then look at the bottom
		 * part (from 0 to cg_irotor).
		 */
		len = start + 1;
		start = 0;
		loc = skpc(0xff, len, &cg_inosused(cgp)[0]);
		if (loc == 0) {
			/*
			 * If we failed again, then either the bitmap or the
			 * counters kept for the cylinder group are wrong.
			 */
			printf("cg = %d, irotor = %d, fs = %s\n",
			    cg, cgp->cg_irotor, fs->fs_fsmnt);
			panic("ffs_nodealloccg: map corrupted");
			/* NOTREACHED */
		}
	}

	/* skpc() returns the position relative to the end */
	i = start + len - loc;

	/*
	 * Okay, so now in 'i' we have the location in the bitmap of a byte
	 * holding a free inode. Find the corresponding bit and set it,
	 * updating cg_irotor as well, accordingly.
	 */
	map = cg_inosused(cgp)[i];
	ipref = i * NBBY;
	for (i = 1; i < (1 << NBBY); i <<= 1, ipref++) {
		if ((map & i) == 0) {
			cgp->cg_irotor = ipref;
			goto gotit;
		}
	}

	printf("fs = %s\n", fs->fs_fsmnt);
	panic("ffs_nodealloccg: block not in map");
	/* NOTREACHED */

gotit:

#ifdef FFS2
	/*
	 * For FFS2, check if all inodes in this cylinder group have been used
	 * at least once. If they haven't, and we are allocating an inode past
	 * the last allocated block of inodes, read in a block and initialize
	 * all inodes in it.
	 */
	if (fs->fs_magic == FS_UFS2_MAGIC &&
	    /* Inode is beyond last initialized block of inodes? */
	    ipref + INOPB(fs) > cgp->cg_initediblk &&
	    /* Has any inode not been used at least once? */
	    cgp->cg_initediblk < cgp->cg_ffs2_niblk) {

		ibp = getblk(ip->i_devvp, fsbtodb(fs,
		    ino_to_fsba(fs, cg * fs->fs_ipg + cgp->cg_initediblk)),
		    (int)fs->fs_bsize, 0, INFSLP);

		memset(ibp->b_data, 0, fs->fs_bsize);
		dp2 = (struct ufs2_dinode *)(ibp->b_data);

		 /* Give each inode a generation number */
		for (i = 0; i < INOPB(fs); i++) {
			while (dp2->di_gen == 0)
```

```c
                                        dp2->di_gen = arc4random();
                                dp2++;
                        }

                         /* Update the counter of initialized inodes */
                        cgp->cg_initediblk += INOPB(fs);
                }
#endif /* FFS2 */

        if (DOINGSOFTDEP(ITOV(ip)))
                softdep_setup_inomapdep(bp, ip, cg * fs->fs_ipg + ipref);

        setbit(cg_inosused(cgp), ipref);

        /* Update the counters we keep on free inodes */
        cgp->cg_cs.cs_nifree--;
        fs->fs_cstotal.cs_nifree--;
        fs->fs_cs(fs, cg).cs_nifree--;
        fs->fs_fmod = 1; /* file system was modified */

        /* Update the counters we keep on allocated directories */
        if ((mode & IFMT) == IFDIR) {
                cgp->cg_cs.cs_ndir++;
                fs->fs_cstotal.cs_ndir++;
                fs->fs_cs(fs, cg).cs_ndir++;
        }

        bdwrite(bp);

#ifdef FFS2
        if (ibp != NULL)
                bawrite(ibp);
#endif

        /* Return the allocated inode number */
        return (cg * fs->fs_ipg + ipref);
}

/*
 * Free a block or fragment.
 *
 * The specified block or fragment is placed back in the
 * free map. If a fragment is deallocated, a possible
 * block reassembly is checked.
 */
void
ffs_blkfree(struct inode *ip, daddr_t bno, long size)
{
        struct fs *fs;
        struct cg *cgp;
        struct buf *bp;
        struct timespec now;
        daddr_t blkno;
        int i, cg, blk, frags, bbase;

        fs = ip->i_fs;
        if ((u_int)size > fs->fs_bsize || fragoff(fs, size) != 0 ||
            fragnum(fs, bno) + numfrags(fs, size) > fs->fs_frag) {
                printf("dev = 0x%x, bsize = %d, size = %ld, fs = %s\n",
                    ip->i_dev, fs->fs_bsize, size, fs->fs_fsmnt);
                panic("ffs_blkfree: bad size");
        }
        cg = dtog(fs, bno);
        if ((u_int)bno >= fs->fs_size) {
                printf("bad block %lld, ino %u\n", (long long)bno,
                    ip->i_number);
                ffs_fserr(fs, DIP(ip, uid), "bad block");
                return;
        }
        if (!(bp = ffs_cgread(fs, ip, cg)))
                return;

        cgp = (struct cg *)bp->b_data;
        nanotime(&now);
        cgp->cg_ffs2_time = now.tv_sec;
        cgp->cg_time = now.tv_sec;

        bno = dtogd(fs, bno);
        if (size == fs->fs_bsize) {
                blkno = fragstoblks(fs, bno);
                if (!ffs_isfreeblock(fs, cg_blksfree(cgp), blkno)) {
                        printf("dev = 0x%x, block = %lld, fs = %s\n",
                            ip->i_dev, (long long)bno, fs->fs_fsmnt);
                        panic("ffs_blkfree: freeing free block");
                }
                ffs_setblock(fs, cg_blksfree(cgp), blkno);
                ffs_clusteracct(fs, cgp, blkno, 1);
                cgp->cg_cs.cs_nbfree++;
                fs->fs_cstotal.cs_nbfree++;
                fs->fs_cs(fs, cg).cs_nbfree++;

                if (fs->fs_magic != FS_UFS2_MAGIC) {
                        i = cbtocylno(fs, bno);
                        cg_blks(fs, cgp, i)[cbtorpos(fs, bno)]++;
                        cg_blktot(cgp)[i]++;
                }

        } else {
                bbase = bno - fragnum(fs, bno);
                /*
                 * decrement the counts associated with the old frags
                 */
                blk = blkmap(fs, cg_blksfree(cgp), bbase);
                ffs_fragacct(fs, blk, cgp->cg_frsum, -1);
                /*
                 * deallocate the fragment
                 */
                frags = numfrags(fs, size);
                for (i = 0; i < frags; i++) {
                        if (isset(cg_blksfree(cgp), bno + i)) {
                                printf("dev = 0x%x, block = %lld, fs = %s\n",
                                    ip->i_dev, (long long)(bno + i),
                                    fs->fs_fsmnt);
                                panic("ffs_blkfree: freeing free frag");
                        }
                        setbit(cg_blksfree(cgp), bno + i);
                }
                cgp->cg_cs.cs_nffree += i;
                fs->fs_cstotal.cs_nffree += i;
                fs->fs_cs(fs, cg).cs_nffree += i;
                /*
                 * add back in counts associated with the new frags
                 */
                blk = blkmap(fs, cg_blksfree(cgp), bbase);
                ffs_fragacct(fs, blk, cgp->cg_frsum, 1);
                /*
                 * if a complete block has been reassembled, account for it
                 */
                blkno = fragstoblks(fs, bbase);
                if (ffs_isblock(fs, cg_blksfree(cgp), blkno)) {
                        cgp->cg_cs.cs_nffree -= fs->fs_frag;
                        fs->fs_cstotal.cs_nffree -= fs->fs_frag;
                        fs->fs_cs(fs, cg).cs_nffree -= fs->fs_frag;
                        ffs_clusteracct(fs, cgp, blkno, 1);
                        cgp->cg_cs.cs_nbfree++;
                        fs->fs_cstotal.cs_nbfree++;
                        fs->fs_cs(fs, cg).cs_nbfree++;

                        if (fs->fs_magic != FS_UFS2_MAGIC) {
                                i = cbtocylno(fs, bbase);
                                cg_blks(fs, cgp, i)[cbtorpos(fs, bbase)]++;
                                cg_blktot(cgp)[i]++;
                        }
                }
        }
        fs->fs_fmod = 1;
        bdwrite(bp);
}

int
ffs_inode_free(struct inode *pip, ufsino_t ino, mode_t mode)
{
        struct vnode *pvp = ITOV(pip);

        if (DOINGSOFTDEP(pvp)) {
                softdep_freefile(pvp, ino, mode);
                return (0);
        }

        return (ffs_freefile(pip, ino, mode));
}

/*
 * Do the actual free operation.
 * The specified inode is placed back in the free map.
 */
int
ffs_freefile(struct inode *pip, ufsino_t ino, mode_t mode)
{
        struct fs *fs;
        struct cg *cgp;
        struct buf *bp;
        struct timespec now;
        u_int cg;

        fs = pip->i_fs;
        if (ino >= fs->fs_ipg * fs->fs_ncg)
                panic("ffs_freefile: range: dev = 0x%x, ino = %d, fs = %s",
                    pip->i_dev, ino, fs->fs_fsmnt);

        cg = ino_to_cg(fs, ino);
        if (!(bp = ffs_cgread(fs, pip, cg)))
                return (0);

        cgp = (struct cg *)bp->b_data;
        nanotime(&now);
        cgp->cg_ffs2_time = now.tv_sec;
        cgp->cg_time = now.tv_sec;

        ino %= fs->fs_ipg;
        if (isclr(cg_inosused(cgp), ino)) {
                printf("dev = 0x%x, ino = %u, fs = %s\n",
                    pip->i_dev, ino, fs->fs_fsmnt);
                if (fs->fs_ronly == 0)
                        panic("ffs_freefile: freeing free inode");
        }
        clrbit(cg_inosused(cgp), ino);
        if (ino < cgp->cg_irotor)
                cgp->cg_irotor = ino;
        cgp->cg_cs.cs_nifree++;
        fs->fs_cstotal.cs_nifree++;
        fs->fs_cs(fs, cg).cs_nifree++;
        if ((mode & IFMT) == IFDIR) {
                cgp->cg_cs.cs_ndir--;
                fs->fs_cstotal.cs_ndir--;
                fs->fs_cs(fs, cg).cs_ndir--;
        }
        fs->fs_fmod = 1;
        bdwrite(bp);
        return (0);
}


/*
 * Find a block of the specified size in the specified cylinder group.
 *
 * It is a panic if a request is made to find a block if none are
 * available.
 */
daddr_t
ffs_mapsearch(struct fs *fs, struct cg *cgp, daddr_t bpref, int allocsiz)
{
        daddr_t bno;
        int start, len, loc, i;
        int blk, field, subfield, pos;

        /*
         * find the fragment by searching through the free block
         * map for an appropriate bit pattern
         */
        if (bpref)
                start = dtogd(fs, bpref) / NBBY;
        else
                start = cgp->cg_frotor / NBBY;
        len = howmany(fs->fs_fpg, NBBY) - start;
        loc = scanc((u_int)len, (u_char *)&cg_blksfree(cgp)[start],
                (u_char *)fragtbl[fs->fs_frag],
                (u_char)(1 << (allocsiz - 1 + (fs->fs_frag % NBBY))));
        if (loc == 0) {
                len = start + 1;
                start = 0;
                loc = scanc((u_int)len, (u_char *)&cg_blksfree(cgp)[0],
                        (u_char *)fragtbl[fs->fs_frag],
                        (u_char)(1 << (allocsiz - 1 + (fs->fs_frag % NBBY))));
                if (loc == 0) {
                        printf("start = %d, len = %d, fs = %s\n",
                            start, len, fs->fs_fsmnt);
                        panic("ffs_alloccg: map corrupted");
                        /* NOTREACHED */
                }
        }
        bno = (start + len - loc) * NBBY;
        cgp->cg_frotor = bno;
        /*
         * found the byte in the map
         * sift through the bits to find the selected frag
         */
        for (i = bno + NBBY; bno < i; bno += fs->fs_frag) {
                blk = blkmap(fs, cg_blksfree(cgp), bno);
                blk <<= 1;
                field = around[allocsiz];
                subfield = inside[allocsiz];
                for (pos = 0; pos <= fs->fs_frag - allocsiz; pos++) {
                        if ((blk & field) == subfield)
                                return (bno + pos);
                        field <<= 1;
                        subfield <<= 1;
                }
        }
        printf("bno = %lld, fs = %s\n", (long long)bno, fs->fs_fsmnt);
        panic("ffs_alloccg: block not in map");
        return (-1);
}

/*
 * Update the cluster map because of an allocation or free.
 *
 * Cnt == 1 means free; cnt == -1 means allocating.
 */
void
ffs_clusteracct(struct fs *fs, struct cg *cgp, daddr_t blkno, int cnt)
{
        int32_t *sump;
        int32_t *lp;
        u_char *freemapp, *mapp;
        int i, start, end, forw, back, map, bit;

        if (fs->fs_contigsumsize <= 0)
                return;
        freemapp = cg_clustersfree(cgp);
        sump = cg_clustersum(cgp);
        /*
         * Allocate or clear the actual block.
         */
        if (cnt > 0)
                setbit(freemapp, blkno);
        else
                clrbit(freemapp, blkno);
        /*
         * Find the size of the cluster going forward.
         */
        start = blkno + 1;
        end = start + fs->fs_contigsumsize;
        if (end >= cgp->cg_nclusterblks)
```

```c
                end = cgp->cg_nclusterblks;
        mapp = &freemapp[start / NBBY];
        map = *mapp++;
        bit = 1 << (start % NBBY);
        for (i = start; i < end; i++) {
                if ((map & bit) == 0)
                        break;
                if ((i & (NBBY - 1)) != (NBBY - 1)) {
                        bit <<= 1;
                } else {
                        map = *mapp++;
                        bit = 1;
                }
        }
        forw = i - start;
        /*
         * Find the size of the cluster going backward.
         */
        start = blkno - 1;
        end = start - fs->fs_contigsumsize;
        if (end < 0)
                end = -1;
        mapp = &freemapp[start / NBBY];
        map = *mapp--;
        bit = 1 << (start % NBBY);
        for (i = start; i > end; i--) {
                if ((map & bit) == 0)
                        break;
                if ((i & (NBBY - 1)) != 0) {
                        bit >>= 1;
                } else {
                        map = *mapp--;
                        bit = 1 << (NBBY - 1);
                }
        }
        back = start - i;
        /*
         * Account for old cluster and the possibly new forward and
         * back clusters.
         */
        i = back + forw + 1;
        if (i > fs->fs_contigsumsize)
                i = fs->fs_contigsumsize;
        sump[i] += cnt;
        if (back > 0)
                sump[back] -= cnt;
        if (forw > 0)
                sump[forw] -= cnt;
        /*
         * Update cluster summary information.
         */
        lp = &sump[fs->fs_contigsumsize];
        for (i = fs->fs_contigsumsize; i > 0; i--)
                if (*lp-- > 0)
                        break;
        fs->fs_maxcluster[cgp->cg_cgx] = i;
}
File: ffs_balloc.c


/*      $OpenBSD: ffs_balloc.c,v 1.45 2019/07/19 00:24:31 cheloha Exp $     */
/*      $NetBSD: ffs_balloc.c,v 1.3 1996/02/09 22:22:21 christos Exp $      */

/*
 * Copyright (c) 2002 Networks Associates Technology, Inc.
 * All rights reserved.
 *
 * This software was developed for the FreeBSD Project by Marshall
 * Kirk McKusick and Network Associates Laboratories, the Security
 * Research Division of Network Associates, Inc. under DARPA/SPAWAR
 * contract N66001-01-C-8035 ("CBOSS"), as part of the DARPA CHATS
 * research program.
 *
 * Copyright (c) 1982, 1986, 1989, 1993
 *      The Regents of the University of California.  All rights reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions
 * are met:
 * 1. Redistributions of source code must retain the above copyright
 *    notice, this list of conditions and the following disclaimer.
 * 2. Redistributions in binary form must reproduce the above copyright
 *    notice, this list of conditions and the following disclaimer in the
 *    documentation and/or other materials provided with the distribution.
 * 3. Neither the name of the University nor the names of its contributors
 *    may be used to endorse or promote products derived from this software
 *    without specific prior written permission.
 *
 * THIS SOFTWARE IS PROVIDED BY THE REGENTS AND CONTRIBUTORS ``AS IS'' AND
 * ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
 * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
 * ARE DISCLAIMED.  IN NO EVENT SHALL THE REGENTS OR CONTRIBUTORS BE LIABLE
 * FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
 * DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
 * OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
 * HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
 * LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
 * OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
 * SUCH DAMAGE.
 *
 *      @(#)ffs_balloc.c 8.4 (Berkeley) 9/23/93
 */

#include <sys/param.h>
#include <sys/systm.h>
#include <sys/buf.h>
#include <sys/proc.h>
#include <sys/mount.h>
#include <sys/vnode.h>

#include <ufs/ufs/quota.h>
#include <ufs/ufs/inode.h>
#include <ufs/ufs/ufsmount.h>
#include <ufs/ufs/ufs_extern.h>

#include <ufs/ffs/fs.h>
#include <ufs/ffs/ffs_extern.h>

int ffs1_balloc(struct inode *, off_t, int, struct ucred *, int, struct buf **);
#ifdef FFS2
int ffs2_balloc(struct inode *, off_t, int, struct ucred *, int, struct buf **);
#endif

/*
 * Balloc defines the structure of file system storage
 * by allocating the physical blocks on a device given
 * the inode and the logical block number in a file.
 */
int
ffs1_balloc(struct inode *ip, off_t startoffset, int size, struct ucred *cred,
    int flags, struct buf **bpp)
{
        daddr_t lbn, nb, newb, pref;
        struct fs *fs;
        struct buf *bp, *nbp;
        struct vnode *vp;
        struct proc *p;
        struct indir indirs[NIADDR + 2];
        int32_t *bap;
        int deallocated, osize, nsize, num, i, error;
        int32_t *allocib, *blkp, *allocblk, allociblk[NIADDR+1];
        int unwindidx = -1;

        vp = ITOV(ip);
        fs = ip->i_fs;
        p = curproc;
        lbn = lblkno(fs, startoffset);
        size = blkoff(fs, startoffset) + size;
        if (size > fs->fs_bsize)
                panic("ffs1_balloc: blk too big");
        if (bpp != NULL)
                *bpp = NULL;
        if (lbn < 0)
                return (EFBIG);

        /*
         * If the next write will extend the file into a new block,
         * and the file is currently composed of a fragment
         * this fragment has to be extended to be a full block.
         */
        nb = lblkno(fs, ip->i_ffs1_size);
        if (nb < NDADDR && nb < lbn) {
                osize = blksize(fs, ip, nb);
                if (osize < fs->fs_bsize && osize > 0) {
                        error = ffs_realloccg(ip, nb,
                            ffs1_blkpref(ip, nb, (int)nb, &ip->i_ffs1_db[0]),
                            osize, (int)fs->fs_bsize, cred, bpp, &newb);
                        if (error)
                                return (error);
                        if (DOINGSOFTDEP(vp))
                                softdep_setup_allocdirect(ip, nb, newb,
                                    ip->i_ffs1_db[nb], fs->fs_bsize, osize,
                                    bpp ? *bpp : NULL);

                        ip->i_ffs1_size = lblktosize(fs, nb + 1);
                        uvm_vnp_setsize(vp, ip->i_ffs1_size);
                        ip->i_ffs1_db[nb] = newb;
                        ip->i_flag |= IN_CHANGE | IN_UPDATE;
                        if (bpp != NULL) {
                                if (flags & B_SYNC)
                                        bwrite(*bpp);
                                else
                                        bawrite(*bpp);
                        }
                }
        }
        /*
         * The first NDADDR blocks are direct blocks
         */
        if (lbn < NDADDR) {
                nb = ip->i_ffs1_db[lbn];
                if (nb != 0 && ip->i_ffs1_size >= lblktosize(fs, lbn + 1)) {
                        /*
                         * The block is an already-allocated direct block
                         * and the file already extends past this block,
                         * thus this must be a whole block.
                         * Just read the block (if requested).
                         */

                        if (bpp != NULL) {
                                error = bread(vp, lbn, fs->fs_bsize, bpp);
                                if (error) {
                                        brelse(*bpp);
                                        return (error);
                                }
                        }
                        return (0);
                }
                if (nb != 0) {
                        /*
                         * Consider need to reallocate a fragment.
                         */
                        osize = fragroundup(fs, blkoff(fs, ip->i_ffs1_size));
                        nsize = fragroundup(fs, size);
                        if (nsize <= osize) {
                                /*
                                 * The existing block is already
                                 * at least as big as we want.
                                 * Just read the block (if requested).
                                 */
                                if (bpp != NULL) {
                                        error = bread(vp, lbn, fs->fs_bsize,
                                            bpp);
                                        if (error) {
                                                brelse(*bpp);
                                                return (error);
                                        }
                                        buf_adjcnt((*bpp), osize);
                                }
                                return (0);
                        } else {
                                /*
                                 * The existing block is smaller than we
                                 * want, grow it.
                                 */
                                error = ffs_realloccg(ip, lbn,
                                    ffs1_blkpref(ip, lbn, (int)lbn,
                                        &ip->i_ffs1_db[0]),
                                    osize, nsize, cred, bpp, &newb);
                                if (error)
                                        return (error);
                                if (DOINGSOFTDEP(vp))
                                        softdep_setup_allocdirect(ip, lbn,
                                            newb, nb, nsize, osize,
                                            bpp ? *bpp : NULL);
                        }
                } else {
                        /*
                         * The block was not previously allocated,
                         * allocate a new block or fragment.
                         */

                        if (ip->i_ffs1_size < lblktosize(fs, lbn + 1))
                                nsize = fragroundup(fs, size);
                        else
                                nsize = fs->fs_bsize;
                        error = ffs_alloc(ip, lbn,
                            ffs1_blkpref(ip, lbn, (int)lbn, &ip->i_ffs1_db[0]),
                            nsize, cred, &newb);
                        if (error)
                                return (error);
                        if (bpp != NULL) {
                                *bpp = getblk(vp, lbn, fs->fs_bsize, 0, INFSLP);
                                if (nsize < fs->fs_bsize)
                                        (*bpp)->b_bcount = nsize;
                                (*bpp)->b_blkno = fsbtodb(fs, newb);
                                if (flags & B_CLRBUF)
                                        clrbuf(*bpp);
                        }
                        if (DOINGSOFTDEP(vp))
                                softdep_setup_allocdirect(ip, lbn, newb, 0,
                                    nsize, 0, bpp ? *bpp : NULL);
                }
                ip->i_ffs1_db[lbn] = newb;
                ip->i_flag |= IN_CHANGE | IN_UPDATE;
                return (0);
        }

        /*
         * Determine the number of levels of indirection.
         */
        pref = 0;
        if ((error = ufs_getlbns(vp, lbn, indirs, &num)) != 0)
                return(error);
#ifdef DIAGNOSTIC
        if (num < 1)
                panic ("ffs1_balloc: ufs_bmaparray returned indirect block");
#endif
        /*
         * Fetch the first indirect block allocating if necessary.
         */
        --num;
        nb = ip->i_ffs1_ib[indirs[0].in_off];

        allocib = NULL;
        allocblk = allociblk;
        if (nb == 0) {
                pref = ffs1_blkpref(ip, lbn, -indirs[0].in_off - 1, NULL);
               error = ffs_alloc(ip, lbn, pref, (int)fs->fs_bsize,
                                 cred, &newb);
                if (error)
                        goto fail;
                nb = newb;
                *allocblk++ = nb;
                bp = getblk(vp, indirs[1].in_lbn, fs->fs_bsize, 0, INFSLP);
                bp->b_blkno = fsbtodb(fs, nb);
                clrbuf(bp);

                if (DOINGSOFTDEP(vp)) {
                        softdep_setup_allocdirect(ip, NDADDR + indirs[0].in_off,
                            newb, 0, fs->fs_bsize, 0, bp);
                        bdwrite(bp);
                } else {
                        /*
                         * Write synchronously so that indirect blocks
                         * never point at garbage.
                         */
                        if ((error = bwrite(bp)) != 0)
                                goto fail;
                }
                allocib = &ip->i_ffs1_ib[indirs[0].in_off];
                *allocib = nb;
                ip->i_flag |= IN_CHANGE | IN_UPDATE;
        }

        /*
         * Fetch through the indirect blocks, allocating as necessary.
         */
        for (i = 1;;) {
                error = bread(vp, indirs[i].in_lbn, (int)fs->fs_bsize, &bp);
                if (error) {
                        brelse(bp);
                        goto fail;
                }
                bap = (int32_t *)bp->b_data;
                nb = bap[indirs[i].in_off];
                if (i == num)
                        break;
                i++;
                if (nb != 0) {
                        brelse(bp);
                        continue;
                }
                if (pref == 0)
                        pref = ffs1_blkpref(ip, lbn, i - num - 1, NULL);
                error = ffs_alloc(ip, lbn, pref, (int)fs->fs_bsize, cred,
                                  &newb);
                if (error) {
                        brelse(bp);
                        goto fail;
                }
                nb = newb;
                *allocblk++ = nb;
                nbp = getblk(vp, indirs[i].in_lbn, fs->fs_bsize, 0, INFSLP);
                nbp->b_blkno = fsbtodb(fs, nb);
                clrbuf(nbp);

                if (DOINGSOFTDEP(vp)) {
                        softdep_setup_allocindir_meta(nbp, ip, bp,
                            indirs[i - 1].in_off, nb);
                        bdwrite(nbp);
                } else {
                        /*
                         * Write synchronously so that indirect blocks
                         * never point at garbage.
                         */
                        if ((error = bwrite(nbp)) != 0) {
                                brelse(bp);
                                goto fail;
                        }
                }
                bap[indirs[i - 1].in_off] = nb;
                if (allocib == NULL && unwindidx < 0)
                        unwindidx = i - 1;
                /*
                 * If required, write synchronously, otherwise use
                 * delayed write.
                 */
                if (flags & B_SYNC) {
                        bwrite(bp);
                } else {
                        bdwrite(bp);
                }
        }
        /*
         * Get the data block, allocating if necessary.
         */
        if (nb == 0) {
                pref = ffs1_blkpref(ip, lbn, indirs[i].in_off, &bap[0]);
                error = ffs_alloc(ip, lbn, pref, (int)fs->fs_bsize, cred,
                                  &newb);
                if (error) {
                        brelse(bp);
                        goto fail;
                }
                nb = newb;
                *allocblk++ = nb;
                if (bpp != NULL) {
                        nbp = getblk(vp, lbn, fs->fs_bsize, 0, INFSLP);
                        nbp->b_blkno = fsbtodb(fs, nb);
                        if (flags & B_CLRBUF)
                                clrbuf(nbp);
                        *bpp = nbp;
                }
                if (DOINGSOFTDEP(vp))
                        softdep_setup_allocindir_page(ip, lbn, bp,
                            indirs[i].in_off, nb, 0, bpp ? *bpp : NULL);
                bap[indirs[i].in_off] = nb;
                /*
                 * If required, write synchronously, otherwise use
                 * delayed write.
                 */
                if (flags & B_SYNC) {
                        bwrite(bp);
                } else {
                        bdwrite(bp);
                }
                return (0);
        }
        brelse(bp);
        if (bpp != NULL) {
                if (flags & B_CLRBUF) {
                        error = bread(vp, lbn, (int)fs->fs_bsize, &nbp);
                        if (error) {
                                brelse(nbp);
                                goto fail;
                        }
                } else {
                        nbp = getblk(vp, lbn, fs->fs_bsize, 0, INFSLP);
                        nbp->b_blkno = fsbtodb(fs, nb);
                }
                *bpp = nbp;
        }
        return (0);

fail:
        /*
         * If we have failed to allocate any blocks, simply return the error.
         * This is the usual case and avoids the need to fsync the file.
         */
        if (allocblk == allociblk && allocib == NULL && unwindidx == -1)
                return (error);
        /*
         * If we have failed part way through block allocation, we have to
         * deallocate any indirect blocks that we have allocated. We have to
         * fsync the file before we start to get rid of all of its
         * dependencies so that we do not leave them dangling. We have to sync
         * it at the end so that the softdep code does not find any untracked
         * changes. Although this is really slow, running out of disk space is
         * not expected to be a common occurrence. The error return from fsync
         * is ignored as we already have an error to return to the user.
         */
        VOP_FSYNC(vp, p->p_ucred, MNT_WAIT, p);
        for (deallocated = 0, blkp = allociblk; blkp < allocblk; blkp++) {
                ffs_blkfree(ip, *blkp, fs->fs_bsize);
                deallocated += fs->fs_bsize;
        }
        if (allocib != NULL) {
                *allocib = 0;
        } else if (unwindidx >= 0) {
                int r;

                r = bread(vp, indirs[unwindidx].in_lbn, (int)fs->fs_bsize, &bp);
                if (r)
                        panic("Could not unwind indirect block, error %d", r);
                bap = (int32_t *)bp->b_data;
                bap[indirs[unwindidx].in_off] = 0;
                if (flags & B_SYNC) {
                        bwrite(bp);
                } else {
                        bdwrite(bp);
                }
        }
        if (deallocated) {
                /*
                 * Restore user's disk quota because allocation failed.
                 */
                (void)ufs_quota_free_blocks(ip, btodb(deallocated), cred);

                ip->i_ffs1_blocks -= btodb(deallocated);
                ip->i_flag |= IN_CHANGE | IN_UPDATE;
        }
        VOP_FSYNC(vp, p->p_ucred, MNT_WAIT, p);
        return (error);
}

#ifdef FFS2
int
ffs2_balloc(struct inode *ip, off_t off, int size, struct ucred *cred,
    int flags, struct buf **bpp)
{
        daddr_t lbn, lastlbn, nb, newb, *blkp;
        daddr_t pref, *allocblk, allociblk[NIADDR + 1];
        daddr_t *bap, *allocib;
        int deallocated, osize, nsize, num, i, error, unwindidx, r;
        struct buf *bp, *nbp;
        struct indir indirs[NIADDR + 2];
        struct fs *fs;
        struct vnode *vp;
        struct proc *p;

        vp = ITOV(ip);
        fs = ip->i_fs;
        p = curproc;
        unwindidx = -1;

        lbn = lblkno(fs, off);
        size = blkoff(fs, off) + size;

        if (size > fs->fs_bsize)
                panic("ffs2_balloc: block too big");

        if (bpp != NULL)
                *bpp = NULL;

        if (lbn < 0)
                return (EFBIG);

        /*
         * If the next write will extend the file into a new block, and the
         * file is currently composed of a fragment, this fragment has to be
         * extended to be a full block.
         */
        lastlbn = lblkno(fs, ip->i_ffs2_size);
        if (lastlbn < NDADDR && lastlbn < lbn) {
                nb = lastlbn;
                osize = blksize(fs, ip, nb);
                if (osize < fs->fs_bsize && osize > 0) {
                        error = ffs_realloccg(ip, nb, ffs2_blkpref(ip,
                            lastlbn, nb, &ip->i_ffs2_db[0]), osize,
                            (int) fs->fs_bsize, cred, bpp, &newb);
                        if (error)
                                return (error);

                        if (DOINGSOFTDEP(vp))
                                softdep_setup_allocdirect(ip, nb, newb,
                                    ip->i_ffs2_db[nb], fs->fs_bsize, osize,
                                    bpp ? *bpp : NULL);

                        ip->i_ffs2_size = lblktosize(fs, nb + 1);
                        uvm_vnp_setsize(vp, ip->i_ffs2_size);
                        ip->i_ffs2_db[nb] = newb;
                        ip->i_flag |= IN_CHANGE | IN_UPDATE;

                        if (bpp) {
                                if (flags & B_SYNC)
                                        bwrite(*bpp);
                                else
                                        bawrite(*bpp);
                        }
                }
        }

        /*
         * The first NDADDR blocks are direct.
         */
        if (lbn < NDADDR) {

                nb = ip->i_ffs2_db[lbn];

                if (nb != 0 && ip->i_ffs2_size >= lblktosize(fs, lbn + 1)) {
                        /*
                         * The direct block is already allocated and the file
                         * extends past this block, thus this must be a whole
                         * block. Just read it, if requested.
                         */
                        if (bpp != NULL) {
                                error = bread(vp, lbn, fs->fs_bsize, bpp);
                                if (error) {
                                        brelse(*bpp);
                                        return (error);
                                }
                        }

                        return (0);
                }

                if (nb != 0) {
                        /*
                         * Consider the need to allocate a fragment.
                         */
                        osize = fragroundup(fs, blkoff(fs, ip->i_ffs2_size));
                        nsize = fragroundup(fs, size);

                        if (nsize <= osize) {
                                /*
                                 * The existing block is already at least as
                                 * big as we want. Just read it, if requested.
                                 */
                                if (bpp != NULL) {
                                        error = bread(vp, lbn, fs->fs_bsize,
                                            bpp);
                                        if (error) {
                                                brelse(*bpp);
                                                return (error);
                                        }
                                        buf_adjcnt((*bpp), osize);
                                }

                                return (0);
                        } else {
                                /*
                                 * The existing block is smaller than we want,
                                 * grow it.
                                 */
                                error = ffs_realloccg(ip, lbn,
                                    ffs2_blkpref(ip, lbn, (int) lbn,
				    &ip->i_ffs2_db[0]), osize, nsize, cred,
				    bpp, &newb);
				if (error)
					return (error);

				if (DOINGSOFTDEP(vp))
					softdep_setup_allocdirect(ip, lbn,
					    newb, nb, nsize, osize,
					    bpp ? *bpp : NULL);
			}
		} else {
			/*
			 * The block was not previously allocated, allocate a
			 * new block or fragment.
			 */
			if (ip->i_ffs2_size < lblktosize(fs, lbn + 1))
				nsize = fragroundup(fs, size);
			else
				nsize = fs->fs_bsize;

			error = ffs_alloc(ip, lbn, ffs2_blkpref(ip, lbn,
			    (int) lbn, &ip->i_ffs2_db[0]), nsize, cred, &newb);
			if (error)
				return (error);

			if (bpp != NULL) {
				bp = getblk(vp, lbn, fs->fs_bsize, 0, INFSLP);
				if (nsize < fs->fs_bsize)
					bp->b_bcount = nsize;
				bp->b_blkno = fsbtodb(fs, newb);
				if (flags & B_CLRBUF)
					clrbuf(bp);
				*bpp = bp;
			}

			if (DOINGSOFTDEP(vp))
				softdep_setup_allocdirect(ip, lbn, newb, 0,
				    nsize, 0, bpp ? *bpp : NULL);
		}

		ip->i_ffs2_db[lbn] = newb;
		ip->i_flag |= IN_CHANGE | IN_UPDATE;

		return (0);
	}

	/*
	 * Determine the number of levels of indirection.
	 */
	pref = 0;
	error = ufs_getlbns(vp, lbn, indirs, &num);
	if (error)
		return (error);

#ifdef DIAGNOSTIC
	if (num < 1)
		panic("ffs2_balloc: ufs_bmaparray returned indirect block");
#endif

	/*
	 * Fetch the first indirect block allocating it necessary.
	 */
	--num;
	nb = ip->i_ffs2_ib[indirs[0].in_off];
	allocib = NULL;
	allocblk = allociblk;

	if (nb == 0) {
		pref = ffs2_blkpref(ip, lbn, -indirs[0].in_off - 1, NULL);
		error = ffs_alloc(ip, lbn, pref, (int) fs->fs_bsize, cred,
		    &newb);
		if (error)
			goto fail;

		nb = newb;
		*allocblk++ = nb;
		bp = getblk(vp, indirs[1].in_lbn, fs->fs_bsize, 0, INFSLP);
		bp->b_blkno = fsbtodb(fs, nb);
		clrbuf(bp);

		if (DOINGSOFTDEP(vp)) {
			softdep_setup_allocdirect(ip, NDADDR + indirs[0].in_off,
			    newb, 0, fs->fs_bsize, 0, bp);
			bdwrite(bp);
		} else {
			/*
			 * Write synchronously so that indirect blocks never
			 * point at garbage.
			 */
			error = bwrite(bp);
			if (error)
				goto fail;
		}

		unwindidx = 0;
		allocib = &ip->i_ffs2_ib[indirs[0].in_off];
		*allocib = nb;
		ip->i_flag |= IN_CHANGE | IN_UPDATE;
	}

	/*
	 * Fetch through the indirect blocks, allocating as necessary.
	 */
	for (i = 1;;) {
		error = bread(vp, indirs[i].in_lbn, (int)fs->fs_bsize, &bp);
		if (error) {
			brelse(bp);
			goto fail;
		}

		bap = (int64_t *) bp->b_data;
		nb = bap[indirs[i].in_off];

		if (i == num)
			break;

		i++;

		if (nb != 0) {
			brelse(bp);
			continue;
		}

		if (pref == 0)
			pref = ffs2_blkpref(ip, lbn, i - num - 1, NULL);

		error = ffs_alloc(ip, lbn, pref, (int) fs->fs_bsize, cred,
		    &newb);
		if (error) {
			brelse(bp);
			goto fail;
		}

		nb = newb;
		*allocblk++ = nb;
		nbp = getblk(vp, indirs[i].in_lbn, fs->fs_bsize, 0, INFSLP);
		nbp->b_blkno = fsbtodb(fs, nb);
		clrbuf(nbp);

		if (DOINGSOFTDEP(vp)) {
			softdep_setup_allocindir_meta(nbp, ip, bp,
			    indirs[i - 1].in_off, nb);
			bdwrite(nbp);
		} else {
			/*
			 * Write synchronously so that indirect blocks never
			 * point at garbage.
			 */
			error = bwrite(nbp);
			if (error) {
				brelse(bp);
				goto fail;
			}
		}

		if (unwindidx < 0)
			unwindidx = i - 1;

		bap[indirs[i - 1].in_off] = nb;

		/*
		 * If required, write synchronously, otherwise use delayed
		 * write.
		 */
		if (flags & B_SYNC)
			bwrite(bp);
		else
			bdwrite(bp);
	}

	/*
	 * Get the data block, allocating if necessary.
	 */
	if (nb == 0) {
		pref = ffs2_blkpref(ip, lbn, indirs[num].in_off, &bap[0]);

		error = ffs_alloc(ip, lbn, pref, (int)fs->fs_bsize, cred,
		    &newb);
		if (error) {
			brelse(bp);
			goto fail;
		}

		nb = newb;
		*allocblk++ = nb;

		if (bpp != NULL) {
			nbp = getblk(vp, lbn, fs->fs_bsize, 0, INFSLP);
			nbp->b_blkno = fsbtodb(fs, nb);
			if (flags & B_CLRBUF)
				clrbuf(nbp);
			*bpp = nbp;
		}

		if (DOINGSOFTDEP(vp))
			softdep_setup_allocindir_page(ip, lbn, bp,
			    indirs[num].in_off, nb, 0, bpp ? *bpp : NULL);

		bap[indirs[num].in_off] = nb;

		if (allocib == NULL && unwindidx < 0)
			unwindidx = i - 1;

		/*
		 * If required, write synchronously, otherwise use delayed
		 * write.
		 */
		if (flags & B_SYNC)
			bwrite(bp);
		else
			bdwrite(bp);

		return (0);
	}

	brelse(bp);

	if (bpp != NULL) {
		if (flags & B_CLRBUF) {
			error = bread(vp, lbn, (int)fs->fs_bsize, &nbp);
			if (error) {
				brelse(nbp);
				goto fail;
			}
		} else {
			nbp = getblk(vp, lbn, fs->fs_bsize, 0, INFSLP);
			nbp->b_blkno = fsbtodb(fs, nb);
			clrbuf(nbp);
		}

		*bpp = nbp;
	}

	return (0);

fail:
	/*
	 * If we have failed to allocate any blocks, simply return the error.
	 * This is the usual case and avoids the need to fsync the file.
	 */
	if (allocblk == allociblk && allocib == NULL && unwindidx == -1)
		return (error);
	/*
	 * If we have failed part way through block allocation, we have to
	 * deallocate any indirect blocks that we have allocated. We have to
	 * fsync the file before we start to get rid of all of its
	 * dependencies so that we do not leave them dangling. We have to sync
	 * it at the end so that the softdep code does not find any untracked
	 * changes. Although this is really slow, running out of disk space is
	 * not expected to be a common occurrence. The error return from fsync
	 * is ignored as we already have an error to return to the user.
	 */
	VOP_FSYNC(vp, p->p_ucred, MNT_WAIT, p);
	if (unwindidx >= 0) {
		/*
		 * First write out any buffers we've created to resolve their
		 * softdeps. This must be done in reverse order of creation so
		 * that we resolve the dependencies in one pass.
		 * Write the cylinder group buffers for these buffers too.
		 */
		 for (i = num; i >= unwindidx; i--) {
			if (i == 0)
				break;

			bp = getblk(vp, indirs[i].in_lbn, (int) fs->fs_bsize,
			    0, INFSLP);
			if (bp->b_flags & B_DELWRI) {
				nb = fsbtodb(fs, cgtod(fs, dtog(fs,
				    dbtofsb(fs, bp->b_blkno))));
				bwrite(bp);
				bp = getblk(ip->i_devvp, nb,
				    (int) fs->fs_cgsize, 0, INFSLP);
				if (bp->b_flags & B_DELWRI)
					bwrite(bp);
				else {
					bp->b_flags |= B_INVAL;
					brelse(bp);
				}
			} else {
				bp->b_flags |= B_INVAL;
				brelse(bp);
			}
		}

		if (DOINGSOFTDEP(vp) && unwindidx == 0) {
			ip->i_flag |= IN_CHANGE | IN_UPDATE;
			ffs_update(ip, 1);
		}

		/*
		 * Now that any dependencies that we created have been
		 * resolved, we can undo the partial allocation.
		 */
		if (unwindidx == 0) {
			*allocib = 0;
			ip->i_flag |= IN_CHANGE | IN_UPDATE;
			if (DOINGSOFTDEP(vp))
				ffs_update(ip, 1);
		} else {
			r = bread(vp, indirs[unwindidx].in_lbn,
			    (int)fs->fs_bsize, &bp);
			if (r)
				panic("ffs2_balloc: unwind failed");
```

```
                        bap = (int64_t *) bp->b_data;
                        bap[indirs[unwindidx].in_off] = 0;
                        bwrite(bp);
                }

                for (i = unwindidx + 1; i <= num; i++) {
                        bp = getblk(vp, indirs[i].in_lbn, (int)fs->fs_bsize, 0,
                            INFSLP);
                        bp->b_flags |= B_INVAL;
                        brelse(bp);
                }
        }

        for (deallocated = 0, blkp = allociblk; blkp < allocblk; blkp++) {
                ffs_blkfree(ip, *blkp, fs->fs_bsize);
                deallocated += fs->fs_bsize;
        }

        if (deallocated) {
                /*
                 * Restore user's disk quota because allocation failed.
                 */
                (void) ufs_quota_free_blocks(ip, btodb(deallocated), cred);

                ip->i_ffs2_blocks -= btodb(deallocated);
                ip->i_flag |= IN_CHANGE | IN_UPDATE;
        }
        VOP_FSYNC(vp, p->p_ucred, MNT_WAIT, p);
        return (error);
}
#endif /* FFS2 */

/*
 * Balloc defines the structure of file system storage by allocating the
 * physical blocks given the inode and the logical block number in a file.
 */
int
ffs_balloc(struct inode *ip, off_t off, int size, struct ucred *cred,
    int flags, struct buf **bpp)
{
#ifdef FFS2
        if (ip->i_fs->fs_magic == FS_UFS2_MAGIC)
                return (ffs2_balloc(ip, off, size, cred, flags, bpp));
        else
#endif
                return (ffs1_balloc(ip, off, size, cred, flags, bpp));
}
File: ffs_extern.h


/*      $OpenBSD: ffs_extern.h,v 1.45 2020/01/20 23:21:56 claudio Exp $     */
/*      $NetBSD: ffs_extern.h,v 1.4 1996/02/09 22:22:22 christos Exp $      */

/*
 * Copyright (c) 1991, 1993, 1994
 *      The Regents of the University of California.  All rights reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions
 * are met:
 * 1. Redistributions of source code must retain the above copyright
 *    notice, this list of conditions and the following disclaimer.
 * 2. Redistributions in binary form must reproduce the above copyright
 *    notice, this list of conditions and the following disclaimer in the
 *    documentation and/or other materials provided with the distribution.
 * 3. Neither the name of the University nor the names of its contributors
 *    may be used to endorse or promote products derived from this software
 *    without specific prior written permission.
 *
 * THIS SOFTWARE IS PROVIDED BY THE REGENTS AND CONTRIBUTORS ``AS IS'' AND
 * ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
 * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
 * ARE DISCLAIMED.  IN NO EVENT SHALL THE REGENTS OR CONTRIBUTORS BE LIABLE
 * FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
 * DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
 * OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
 * HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
 * LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
 * OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
 * SUCH DAMAGE.
 *
 *      @(#)ffs_extern.h 8.3 (Berkeley) 4/16/94
 */

#define FFS_CLUSTERREAD         1       /* cluster reading enabled */
#define FFS_CLUSTERWRITE 2      /* cluster writing enabled */
#define FFS_REALLOCBLKS         3       /* block reallocation enabled */
#define FFS_ASYNCFREE           4       /* asynchronous block freeing enabled */
#define FFS_MAX_SOFTDEPS 5      /* maximum structs before slowdown */
#define FFS_SD_TICKDELAY 6      /* ticks to pause during slowdown */
#define FFS_SD_WORKLIST_PUSH    7       /* # of worklist cleanups */
#define FFS_SD_BLK_LIMIT_PUSH   8       /* # of times block limit neared */
#define FFS_SD_INO_LIMIT_PUSH   9       /* # of times inode limit neared */
#define FFS_SD_BLK_LIMIT_HIT    10      /* # of times block slowdown imposed */
#define FFS_SD_INO_LIMIT_HIT    11      /* # of times inode slowdown imposed */
#define FFS_SD_SYNC_LIMIT_HIT   12      /* # of synchronous slowdowns imposed */
#define FFS_SD_INDIR_BLK_PTRS   13      /* bufs redirtied as indir ptrs not written */
#define FFS_SD_INODE_BITMAP     14      /* bufs redirtied as inode bitmap not written */
#define FFS_SD_DIRECT_BLK_PTRS  15      /* bufs redirtied as direct ptrs not written */
#define FFS_SD_DIR_ENTRY 16     /* bufs redirtied as dir entry cannot write */
#define FFS_DIRHASH_DIRSIZE     17      /* min directory size, in bytes */
#define FFS_DIRHASH_MAXMEM      18      /* max kvm to use, in bytes */
#define FFS_DIRHASH_MEM         19      /* current mem usage, in bytes */
#define FFS_MAXID               20      /* number of valid ffs ids */

#define FFS_NAMES { \
        { 0, 0 }, \
        { 0, 0 }, \
        { 0, 0 }, \
        { 0, 0 }, \
        { 0, 0 }, \
        { "max_softdeps", CTLTYPE_INT }, \
        { "sd_tickdelay", CTLTYPE_INT }, \
        { "sd_worklist_push", CTLTYPE_INT }, \
        { "sd_blk_limit_push", CTLTYPE_INT }, \
        { "sd_ino_limit_push", CTLTYPE_INT }, \
        { "sd_blk_limit_hit", CTLTYPE_INT }, \
        { "sd_ino_limit_hit", CTLTYPE_INT }, \
        { "sd_sync_limit_hit", CTLTYPE_INT }, \
        { "sd_indir_blk_ptrs", CTLTYPE_INT }, \
        { "sd_inode_bitmap", CTLTYPE_INT }, \
        { "sd_direct_blk_ptrs", CTLTYPE_INT }, \
        { "sd_dir_entry", CTLTYPE_INT }, \
        { "dirhash_dirsize", CTLTYPE_INT }, \
        { "dirhash_maxmem", CTLTYPE_INT }, \
        { "dirhash_mem", CTLTYPE_INT }, \
}

struct buf;
struct fid;
struct fs;
struct inode;
struct mount;
struct nameidata;
struct proc;
struct statfs;
struct timeval;
struct ucred;
struct ufsmount;
struct vfsconf;
struct uio;
struct vnode;
struct mbuf;
struct cg;
struct vop_vfree_args;

extern const struct vops  ffs_vops;
extern const struct vops  ffs_specvops;
extern const struct vops  ffs_fifovops;

/* ffs_alloc.c */
int ffs_alloc(struct inode *, daddr_t, daddr_t , int, struct ucred *,
                    daddr_t *);
int ffs_reallocccg(struct inode *, daddr_t, daddr_t, int, int ,
                        struct ucred *, struct buf **, daddr_t *);
int ffs_inode_alloc(struct inode *, mode_t, struct ucred *, struct vnode **);
int ffs_inode_free(struct inode *, ufsino_t, mode_t);
int ffs_freefile(struct inode *, ufsino_t, mode_t);

int32_t ffs1_blkpref(struct inode *, daddr_t, int, int32_t *);
#ifdef FFS2
int64_t ffs2_blkpref(struct inode *, daddr_t, int, int64_t *);
#endif
void ffs_blkfree(struct inode *, daddr_t, long);
void ffs_clusteracct(struct fs *, struct cg *, daddr_t, int);

/* ffs_balloc.c */
int ffs_balloc(struct inode *, off_t, int, struct ucred *, int, struct buf **);

/* ffs_inode.c */
int ffs_init(struct vfsconf *);
int ffs_update(struct inode *, int);
int ffs_truncate(struct inode *, off_t, int, struct ucred *);

/* ffs_subr.c */
int  ffs_bufatoff(struct inode *, off_t, char **, struct buf **);
void ffs_fragacct(struct fs *, int, int32_t[], int);
#ifdef DIAGNOSTIC
void    ffs_checkoverlap(struct buf *, struct inode *);
#endif
int  ffs_isfreeblock(struct fs *, u_char *, daddr_t);
int  ffs_isblock(struct fs *, u_char *, daddr_t);
void ffs_clrblock(struct fs *, u_char *, daddr_t);
void ffs_setblock(struct fs *, u_char *, daddr_t);
int  ffs_vinit(struct mount *, struct vnode **);

/* ffs_vfsops.c */
int ffs_mountroot(void);
int ffs_mount(struct mount *, const char *, void *, struct nameidata *,
                    struct proc *);
int ffs_reload(struct mount *, struct ucred *, struct proc *);
int ffs_mountfs(struct vnode *, struct mount *, struct proc *);
int ffs_oldfscompat(struct fs *);
int ffs_unmount(struct mount *, int, struct proc *);
int ffs_flushfiles(struct mount *, int, struct proc *);
int ffs_statfs(struct mount *, struct statfs *, struct proc *);
int ffs_sync(struct mount *, int, int, struct ucred *, struct proc *);
int ffs_vget(struct mount *, ino_t, struct vnode **);
int ffs_fhtovp(struct mount *, struct fid *, struct vnode **);
int ffs_vptofh(struct vnode *, struct fid *);
int ffs_sysctl(int *, u_int, void *, size_t *, void *, size_t,
                     struct proc *);
int ffs_sbupdate(struct ufsmount *, int);
int ffs_cgupdate(struct ufsmount *, int);

/* ffs_vnops.c */
int ffs_read(void *);
int ffs_write(void *);
int ffs_fsync(void *);
int ffs_reclaim(void *);
int ffsfifo_reclaim(void *);

/*
 * Soft dependency function prototypes.
 */

struct vop_vfree_args;
struct vop_fsync_args;

void  softdep_initialize(void);
int   softdep_process_worklist(struct mount *);
int   softdep_mount(struct vnode *, struct mount *, struct fs *,
          struct ucred *);
int   softdep_flushworklist(struct mount *, int *, struct proc *);
int   softdep_flushfiles(struct mount *, int, struct proc *);
void  softdep_update_inodeblock(struct inode *, struct buf *, int);
void  softdep_load_inodeblock(struct inode *);
void  softdep_freefile(struct vnode *, ufsino_t, mode_t);
void  softdep_setup_freeblocks(struct inode *, off_t);
void  softdep_setup_inomapdep(struct buf *, struct inode *, ufsino_t);
void  softdep_setup_blkmapdep(struct buf *, struct fs *, daddr_t);
void  softdep_setup_allocdirect(struct inode *, daddr_t, daddr_t,
            daddr_t, long, long, struct buf *);
void  softdep_setup_allocindir_meta(struct buf *, struct inode *,
            struct buf *, int, daddr_t);
void  softdep_setup_allocindir_page(struct inode *, daddr_t,
            struct buf *, int, daddr_t, daddr_t, struct buf *);
void  softdep_fsync_mountdev(struct vnode *, int);
int   softdep_sync_metadata(struct vop_fsync_args *);
int   softdep_fsync(struct vnode *);

extern struct pool ffs_ino_pool;  /* memory pool for inodes */
extern struct pool ffs_dinode1_pool;      /* memory pool for UFS1 dinodes */
#ifdef FFS2
extern struct pool ffs_dinode2_pool;      /* memory pool for UFS2 dinodes */
#endif
File: ffs_inode.c


/*      $OpenBSD: ffs_inode.c,v 1.80 2019/07/25 01:43:21 cheloha Exp $      */
/*      $NetBSD: ffs_inode.c,v 1.10 1996/05/11 18:27:19 mycroft Exp $       */

/*
 * Copyright (c) 1982, 1986, 1989, 1993
 *      The Regents of the University of California.  All rights reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions
 * are met:
 * 1. Redistributions of source code must retain the above copyright
 *    notice, this list of conditions and the following disclaimer.
 * 2. Redistributions in binary form must reproduce the above copyright
 *    notice, this list of conditions and the following disclaimer in the
 *    documentation and/or other materials provided with the distribution.
 * 3. Neither the name of the University nor the names of its contributors
 *    may be used to endorse or promote products derived from this software
 *    without specific prior written permission.
 *
 * THIS SOFTWARE IS PROVIDED BY THE REGENTS AND CONTRIBUTORS ``AS IS'' AND
 * ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
 * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
 * ARE DISCLAIMED.  IN NO EVENT SHALL THE REGENTS OR CONTRIBUTORS BE LIABLE
 * FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
 * DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
 * OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
 * HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
 * LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
 * OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
 * SUCH DAMAGE.
 *
 *      @(#)ffs_inode.c  8.8 (Berkeley) 10/19/94
 */

#include <sys/param.h>
#include <sys/systm.h>
#include <sys/mount.h>
#include <sys/proc.h>
#include <sys/buf.h>
#include <sys/vnode.h>
#include <sys/kernel.h>
#include <sys/malloc.h>
#include <sys/resourcevar.h>

#include <ufs/ufs/quota.h>
#include <ufs/ufs/inode.h>
#include <ufs/ufs/ufsmount.h>
#include <ufs/ufs/ufs_extern.h>

#include <ufs/ffs/fs.h>
#include <ufs/ffs/ffs_extern.h>

int ffs_indirtrunc(struct inode *, daddr_t, daddr_t, daddr_t, int, long *);

/*
```

```c
 * Update the access, modified, and inode change times as specified by the
 * IN_ACCESS, IN_UPDATE, and IN_CHANGE flags respectively. The IN_MODIFIED
 * flag is used to specify that the inode needs to be updated but that the
 * times have already been set.  The IN_LAZYMOD flag is used to specify
 * that the inode needs to be updated at some point, by reclaim if not
 * in the course of other changes; this is used to defer writes just to
 * update device timestamps.  If waitfor is set, then wait for the disk
 * write of the inode to complete.
 */
int
ffs_update(struct inode *ip, int waitfor)
{
        struct vnode *vp;
        struct fs *fs;
        struct buf *bp;
        int error;

        vp = ITOV(ip);
        ufs_itimes(vp);

        if ((ip->i_flag & IN_MODIFIED) == 0 && waitfor == 0)
                return (0);

        ip->i_flag &= ~(IN_MODIFIED | IN_LAZYMOD);
        fs = ip->i_fs;

        /*
         * Ensure that uid and gid are correct. This is a temporary
         * fix until fsck has been changed to do the update.
         */
        if (fs->fs_magic == FS_UFS1_MAGIC && fs->fs_inodefmt < FS_44INODEFMT) {
                ip->i_din1->di_ouid = ip->i_ffs1_uid;
                ip->i_din1->di_ogid = ip->i_ffs1_gid;
        }

        error = bread(ip->i_devvp, fsbtodb(fs, ino_to_fsba(fs, ip->i_number)),
            (int)fs->fs_bsize, &bp);
        if (error) {
                brelse(bp);
                return (error);
        }

        if (DOINGSOFTDEP(vp))
                softdep_update_inodeblock(ip, bp, waitfor);
        else if (ip->i_effnlink != DIP(ip, nlink))
                panic("ffs_update: bad link cnt");

#ifdef FFS2
        if (ip->i_ump->um_fstype == UM_UFS2)
                *((struct ufs2_dinode *)bp->b_data +
                    ino_to_fsbo(fs, ip->i_number)) = *ip->i_din2;
        else
#endif
                *((struct ufs1_dinode *)bp->b_data +
                    ino_to_fsbo(fs, ip->i_number)) = *ip->i_din1;

        if (waitfor && !DOINGASYNC(vp)) {
                return (bwrite(bp));
        } else {
                bdwrite(bp);
                return (0);
        }
}

#define SINGLE  0       /* index of single indirect block */
#define DOUBLE  1       /* index of double indirect block */
#define TRIPLE  2       /* index of triple indirect block */

/*
 * Truncate the inode oip to at most length size, freeing the
 * disk blocks.
 */
int
ffs_truncate(struct inode *oip, off_t length, int flags, struct ucred *cred)
{
        struct vnode *ovp;
        daddr_t lastblock;
        daddr_t bn, lbn, lastiblock[NIADDR], indir_lbn[NIADDR];
        daddr_t oldblks[NDADDR + NIADDR], newblks[NDADDR + NIADDR];
        struct fs *fs;
        struct buf *bp;
        int offset, size, level;
        long count, nblocks, vflags, blocksreleased = 0;
        int i, aflags, error, allerror;
        off_t osize;

        if (length < 0)
                return (EINVAL);
        ovp = ITOV(oip);

        if (ovp->v_type != VREG &&
            ovp->v_type != VDIR &&
            ovp->v_type != VLNK)
                return (0);

        if (DIP(oip, size) == length)
                return (0);

        if (ovp->v_type == VLNK &&
            (DIP(oip, size) < oip->i_ump->um_maxsymlinklen ||
             (oip->i_ump->um_maxsymlinklen == 0 &&
              oip->i_din1->di_blocks == 0))) {
#ifdef DIAGNOSTIC
                if (length != 0)
                        panic("ffs_truncate: partial truncate of symlink");
#endif
                memset(SHORTLINK(oip), 0, (size_t) DIP(oip, size));
                DIP_ASSIGN(oip, size, 0);
                oip->i_flag |= IN_CHANGE | IN_UPDATE;
                return (UFS_UPDATE(oip, 1));
        }

        if ((error = getinoquota(oip)) != 0)
                return (error);

        fs = oip->i_fs;
        if (length > fs->fs_maxfilesize)
                return (EFBIG);

        uvm_vnp_setsize(ovp, length);
        oip->i_ci.ci_lasta = oip->i_ci.ci_clen
            = oip->i_ci.ci_cstart = oip->i_ci.ci_lastw = 0;

        if (DOINGSOFTDEP(ovp)) {
                if (length > 0 || softdep_slowdown(ovp)) {
                        /*
                         * If a file is only partially truncated, then
                         * we have to clean up the data structures
                         * describing the allocation past the truncation
                         * point. Finding and deallocating those structures
                         * is a lot of work. Since partial truncation occurs
                         * rarely, we solve the problem by syncing the file
                         * so that it will have no data structures left.
                         */
                        if ((error = VOP_FSYNC(ovp, cred, MNT_WAIT,
                                                curproc)) != 0)
                                return (error);
                } else {
                        (void)ufs_quota_free_blocks(oip, DIP(oip, blocks),
                            NOCRED);
                        softdep_setup_freeblocks(oip, length);
                        vinvalbuf(ovp, 0, cred, curproc, 0, INFSLP);
                        oip->i_flag |= IN_CHANGE | IN_UPDATE;
                        return (UFS_UPDATE(oip, 0));
                }
        }

        osize = DIP(oip, size);
        /*
         * Lengthen the size of the file. We must ensure that the
         * last byte of the file is allocated. Since the smallest
         * value of osize is 0, length will be at least 1.
         */
        if (osize < length) {
                aflags = B_CLRBUF;
                if (flags & IO_SYNC)
                        aflags |= B_SYNC;
                error = UFS_BUF_ALLOC(oip, length - 1, 1,
                                   cred, aflags, &bp);
                if (error)
                        return (error);
                DIP_ASSIGN(oip, size, length);
                uvm_vnp_setsize(ovp, length);
                (void) uvm_vnp_uncache(ovp);
                if (aflags & B_SYNC)
                        bwrite(bp);
                else
                        bawrite(bp);
                oip->i_flag |= IN_CHANGE | IN_UPDATE;
                return (UFS_UPDATE(oip, 1));
        }
        uvm_vnp_setsize(ovp, length);

        /*
         * Shorten the size of the file. If the file is not being
         * truncated to a block boundary, the contents of the
         * partial block following the end of the file must be
         * zero'ed in case it ever becomes accessible again because
         * of subsequent file growth. Directories however are not
         * zero'ed as they should grow back initialized to empty.
         */
        offset = blkoff(fs, length);
        if (offset == 0) {
                DIP_ASSIGN(oip, size, length);
        } else {
                lbn = lblkno(fs, length);
                aflags = B_CLRBUF;
                if (flags & IO_SYNC)
                        aflags |= B_SYNC;
                error = UFS_BUF_ALLOC(oip, length - 1, 1,
                                   cred, aflags, &bp);
                if (error)
                        return (error);
                /*
                 * When we are doing soft updates and the UFS_BALLOC
                 * above fills in a direct block hole with a full sized
                 * block that will be truncated down to a fragment below,
                 * we must flush out the block dependency with an FSYNC
                 * so that we do not get a soft updates inconsistency
                 * when we create the fragment below.
                 */
                if (DOINGSOFTDEP(ovp) && lbn < NDADDR &&
                    fragroundup(fs, blkoff(fs, length)) < fs->fs_bsize &&
                    (error = VOP_FSYNC(ovp, cred, MNT_WAIT, curproc)) != 0)
                        return (error);
                DIP_ASSIGN(oip, size, length);
                size = blksize(fs, oip, lbn);
                (void) uvm_vnp_uncache(ovp);
                if (ovp->v_type != VDIR)
                        memset(bp->b_data + offset, 0, size - offset);
                buf_adjcnt(bp, size);
                if (aflags & B_SYNC)
                        bwrite(bp);
                else
                        bawrite(bp);
        }
        /*
         * Calculate index into inode's block list of
         * last direct and indirect blocks (if any)
         * which we want to keep.  Lastblock is -1 when
         * the file is truncated to 0.
         */
        lastblock = lblkno(fs, length + fs->fs_bsize - 1) - 1;
        lastiblock[SINGLE] = lastblock - NDADDR;
        lastiblock[DOUBLE] = lastiblock[SINGLE] - NINDIR(fs);
        lastiblock[TRIPLE] = lastiblock[DOUBLE] - NINDIR(fs) * NINDIR(fs);
        nblocks = btodb(fs->fs_bsize);

        /*
         * Update file and block pointers on disk before we start freeing
         * blocks.  If we crash before free'ing blocks below, the blocks
         * will be returned to the free list.  lastiblock values are also
         * normalized to -1 for calls to ffs_indirtrunc below.
         */
        for (level = TRIPLE; level >= SINGLE; level--) {
                oldblks[NDADDR + level] = DIP(oip, ib[level]);
                if (lastiblock[level] < 0) {
                        DIP_ASSIGN(oip, ib[level], 0);
                        lastiblock[level] = -1;
                }
        }

        for (i = 0; i < NDADDR; i++) {
                oldblks[i] = DIP(oip, db[i]);
                if (i > lastblock)
                        DIP_ASSIGN(oip, db[i], 0);
        }

        oip->i_flag |= IN_CHANGE | IN_UPDATE;
        if ((error = UFS_UPDATE(oip, 1)) != 0)
                allerror = error;

        /*
         * Having written the new inode to disk, save its new configuration
         * and put back the old block pointers long enough to process them.
         * Note that we save the new block configuration so we can check it
         * when we are done.
         */
        for (i = 0; i < NDADDR; i++) {
                newblks[i] = DIP(oip, db[i]);
                DIP_ASSIGN(oip, db[i], oldblks[i]);
        }

        for (i = 0; i < NIADDR; i++) {
                newblks[NDADDR + i] = DIP(oip, ib[i]);
                DIP_ASSIGN(oip, ib[i], oldblks[NDADDR + i]);
        }

        DIP_ASSIGN(oip, size, osize);
        vflags = ((length > 0) ? V_SAVE : 0) | V_SAVEMETA;
        allerror = vinvalbuf(ovp, vflags, cred, curproc, 0, INFSLP);

        /*
         * Indirect blocks first.
         */
        indir_lbn[SINGLE] = -NDADDR;
        indir_lbn[DOUBLE] = indir_lbn[SINGLE] - NINDIR(fs) - 1;
        indir_lbn[TRIPLE] = indir_lbn[DOUBLE] - NINDIR(fs) * NINDIR(fs) - 1;
        for (level = TRIPLE; level >= SINGLE; level--) {
                bn = DIP(oip, ib[level]);
                if (bn != 0) {
                        error = ffs_indirtrunc(oip, indir_lbn[level],
                            fsbtodb(fs, bn), lastiblock[level], level, &count);
                        if (error)
                                allerror = error;
                        blocksreleased += count;
                        if (lastiblock[level] < 0) {
                                DIP_ASSIGN(oip, ib[level], 0);
                                ffs_blkfree(oip, bn, fs->fs_bsize);
                                blocksreleased += nblocks;
                        }
                }
                if (lastiblock[level] >= 0)
                        goto done;
        }

        /*
         * All whole direct blocks or frags.
         */
        for (i = NDADDR - 1; i > lastblock; i--) {
                long bsize;

                bn = DIP(oip, db[i]);
                if (bn == 0)
```

```
                        continue;

                DIP_ASSIGN(oip, db[i], 0);
                bsize = blksize(fs, oip, i);
                ffs_blkfree(oip, bn, bsize);
                blocksreleased += btodb(bsize);
        }
        if (lastblock < 0)
                goto done;

        /*
         * Finally, look for a change in size of the
         * last direct block; release any frags.
         */
        bn = DIP(oip, db[lastblock]);
        if (bn != 0) {
                long oldspace, newspace;

                /*
                 * Calculate amount of space we're giving
                 * back as old block size minus new block size.
                 */
                oldspace = blksize(fs, oip, lastblock);
                DIP_ASSIGN(oip, size, length);
                newspace = blksize(fs, oip, lastblock);
                if (newspace == 0)
                        panic("ffs_truncate: newspace");
                if (oldspace - newspace > 0) {
                        /*
                         * Block number of space to be free'd is
                         * the old block # plus the number of frags
                         * required for the storage we're keeping.
                         */
                        bn += numfrags(fs, newspace);
                        ffs_blkfree(oip, bn, oldspace - newspace);
                        blocksreleased += btodb(oldspace - newspace);
                }
        }
done:
#ifdef DIAGNOSTIC
        for (level = SINGLE; level <= TRIPLE; level++)
                if (newblks[NDADDR + level] != DIP(oip, ib[level]))
                        panic("ffs_truncate1");
        for (i = 0; i < NDADDR; i++)
                if (newblks[i] != DIP(oip, db[i]))
                        panic("ffs_truncate2");
#endif /* DIAGNOSTIC */
        /*
         * Put back the real size.
         */
        DIP_ASSIGN(oip, size, length);
        if (DIP(oip, blocks) >= blocksreleased)
                DIP_ADD(oip, blocks, -blocksreleased);
        else    /* sanity */
                DIP_ASSIGN(oip, blocks, 0);
        oip->i_flag |= IN_CHANGE;
        (void)ufs_quota_free_blocks(oip, blocksreleased, NOCRED);
        return (allerror);
}

#ifdef FFS2
#define BAP(ip, i) (((ip)->i_ump->um_fstype == UM_UFS2) ? bap2[i] : bap1[i])
#define BAP_ASSIGN(ip, i, value)                                        \
        do {                                                            \
                if ((ip)->i_ump->um_fstype == UM_UFS2)                  \
                        bap2[i] = (value);                              \
                else                                                    \
                        bap1[i] = (value);                              \
        } while (0)
#else
#define BAP(ip, i) bap1[i]
#define BAP_ASSIGN(ip, i, value) do { bap1[i] = (value); } while (0)
#endif /* FFS2 */

/*
 * Release blocks associated with the inode ip and stored in the indirect
 * block bn.  Blocks are free'd in LIFO order up to (but not including)
 * lastbn.  If level is greater than SINGLE, the block is an indirect block
 * and recursive calls to indirtrunc must be used to cleanse other indirect
 * blocks.
 *
 * NB: triple indirect blocks are untested.
 */
int
ffs_indirtrunc(struct inode *ip, daddr_t lbn, daddr_t dbn,
    daddr_t lastbn, int level, long *countp)
{
        int i;
        struct buf *bp;
        struct fs *fs = ip->i_fs;
        struct vnode *vp;
        void *copy = NULL;
        daddr_t nb, nlbn, last;
        long blkcount, factor;
        int nblocks, blocksreleased = 0;
        int error = 0, allerror = 0;
        int32_t *bap1 = NULL;
#ifdef FFS2
        int64_t *bap2 = NULL;
#endif

        /*
         * Calculate index in current block of last
         * block to be kept.  -1 indicates the entire
         * block so we need not calculate the index.
         */
        factor = 1;
        for (i = SINGLE; i < level; i++)
                factor *= NINDIR(fs);
        last = lastbn;
        if (lastbn > 0)
                last /= factor;
        nblocks = btodb(fs->fs_bsize);
        /*
         * Get buffer of block pointers, zero those entries corresponding
         * to blocks to be free'd, and update on disk copy first.  Since
         * double(triple) indirect before single(double) indirect, calls
         * to bmap on these blocks will fail.  However, we already have
         * the on disk address, so we have to set the b_blkno field
         * explicitly instead of letting bread do everything for us.
         */
        vp = ITOV(ip);
        bp = getblk(vp, lbn, (int)fs->fs_bsize, 0, INFSLP);
        if (!(bp->b_flags & (B_DONE | B_DELWRI))) {
                curproc->p_ru.ru_inblock++;             /* pay for read */
                bcstats.pendingreads++;
                bcstats.numreads++;
                bp->b_flags |= B_READ;
                if (bp->b_bcount > bp->b_bufsize)
                        panic("ffs_indirtrunc: bad buffer size");
                bp->b_blkno = dbn;
                VOP_STRATEGY(bp);
                error = biowait(bp);
        }
        if (error) {
                brelse(bp);
                *countp = 0;
                return (error);
        }

#ifdef FFS2
        if (ip->i_ump->um_fstype == UM_UFS2)
                bap2 = (int64_t *)bp->b_data;
        else
#endif
                bap1 = (int32_t *)bp->b_data;

        if (lastbn != -1) {
                copy = malloc(fs->fs_bsize, M_TEMP, M_WAITOK);
                memcpy(copy, bp->b_data, fs->fs_bsize);

                for (i = last + 1; i < NINDIR(fs); i++)
                        BAP_ASSIGN(ip, i, 0);

                if (!DOINGASYNC(vp)) {
                        error = bwrite(bp);
                        if (error)
                                allerror = error;
                } else {
                        bawrite(bp);
                }

#ifdef FFS2
                if (ip->i_ump->um_fstype == UM_UFS2)
                        bap2 = (int64_t *)copy;
                else
#endif
                        bap1 = (int32_t *)copy;
        }

        /*
         * Recursively free totally unused blocks.
         */
        for (i = NINDIR(fs) - 1, nlbn = lbn + 1 - i * factor; i > last;
            i--, nlbn += factor) {
                nb = BAP(ip, i);
                if (nb == 0)
                        continue;
                if (level > SINGLE) {
                        error = ffs_indirtrunc(ip, nlbn, fsbtodb(fs, nb),
                                               -1, level - 1, &blkcount);
                        if (error)
                                allerror = error;
                        blocksreleased += blkcount;
                }
                ffs_blkfree(ip, nb, fs->fs_bsize);
                blocksreleased += nblocks;
        }

        /*
         * Recursively free last partial block.
         */
        if (level > SINGLE && lastbn >= 0) {
                last = lastbn % factor;
                nb = BAP(ip, i);
                if (nb != 0) {
                        error = ffs_indirtrunc(ip, nlbn, fsbtodb(fs, nb),
                                               last, level - 1, &blkcount);
                        if (error)
                                allerror = error;
                        blocksreleased += blkcount;
                }
        }
        if (copy != NULL) {
                free(copy, M_TEMP, fs->fs_bsize);
        } else {
                bp->b_flags |= B_INVAL;
                brelse(bp);
        }

        *countp = blocksreleased;
        return (allerror);
}
File: ffs_softdep.c


/*      $OpenBSD: ffs_softdep.c,v 1.149 2020/03/09 16:49:12 millert Exp $   */

/*
 * Copyright 1998, 2000 Marshall Kirk McKusick. All Rights Reserved.
 *
 * The soft updates code is derived from the appendix of a University
 * of Michigan technical report (Gregory R. Ganger and Yale N. Patt,
 * "Soft Updates: A Solution to the Metadata Update Problem in File
 * Systems", CSE-TR-254-95, August 1995).
 *
 * Further information about soft updates can be obtained from:
 *
 *      Marshall Kirk McKusick          http://www.mckusick.com/softdep/
 *      1614 Oxford Street              mckusick@mckusick.com
 *      Berkeley, CA 94709-1608         +1-510-843-9542
 *      USA
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions
 * are met:
 *
 * 1. Redistributions of source code must retain the above copyright
 *    notice, this list of conditions and the following disclaimer.
 * 2. Redistributions in binary form must reproduce the above copyright
 *    notice, this list of conditions and the following disclaimer in the
 *    documentation and/or other materials provided with the distribution.
 *
 * THIS SOFTWARE IS PROVIDED BY MARSHALL KIRK MCKUSICK ``AS IS'' AND ANY
 * EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED
 * WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE
 * DISCLAIMED.  IN NO EVENT SHALL MARSHALL KIRK MCKUSICK BE LIABLE FOR
 * ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
 * DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
 * OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
 * HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
 * LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
 * OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
 * SUCH DAMAGE.
 *
 *      from: @(#)ffs_softdep.c  9.59 (McKusick) 6/21/00
 * $FreeBSD: src/sys/ufs/ffs/ffs_softdep.c,v 1.86 2001/02/04 16:08:18 phk Exp $
 */

#include <sys/param.h>
#include <sys/buf.h>
#include <sys/kernel.h>
#include <sys/malloc.h>
#include <sys/mount.h>
#include <sys/proc.h>
#include <sys/pool.h>
#include <sys/syslog.h>
#include <sys/systm.h>
#include <sys/vnode.h>
#include <sys/specdev.h>
#include <crypto/siphash.h>
#include <ufs/ufs/dir.h>
#include <ufs/ufs/quota.h>
#include <ufs/ufs/inode.h>
#include <ufs/ufs/ufsmount.h>
#include <ufs/ffs/fs.h>
#include <ufs/ffs/softdep.h>
#include <ufs/ffs/ffs_extern.h>
#include <ufs/ufs/ufs_extern.h>

#define STATIC

/*
 * Mapping of dependency structure types to malloc types.
 */
#define D_PAGEDEP       0
#define D_INODEDEP      1
#define D_NEWBLK 2
#define D_BMSAFEMAP     3
#define D_ALLOCDIRECT   4
#define D_INDIRDEP      5
#define D_ALLOCINDIR    6
#define D_FREEFRAG      7
#define D_FREEBLKS      8
#define D_FREEFILE      9
#define D_DIRADD 10
#define D_MKDIR         11
#define D_DIRREM 12
#define D_NEWDIRBLK     13
#define D_LAST          13
/*
 * Names of softdep types.
 */
const char *softdep_typenames[] = {
	"pagedep",
	"inodedep",
	"newblk",
	"bmsafemap",
	"allocdirect",
	"indirdep",
	"allocindir",
	"freefrag",
	"freeblks",
	"freefile",
	"diradd",
	"mkdir",
	"dirrem",
	"newdirblk",
};
#define	TYPENAME(type) \
	((unsigned)(type) <= D_LAST ? softdep_typenames[type] : "???")
/*
 * Finding the current process.
 */
#define CURPROC curproc
/*
 * End system adaptation definitions.
 */

/*
 * Internal function prototypes.
 */
STATIC	void softdep_error(char *, int);
STATIC	void drain_output(struct vnode *, int);
STATIC	int getdirtybuf(struct buf *, int);
STATIC	void clear_remove(struct proc *);
STATIC	void clear_inodedeps(struct proc *);
STATIC	int flush_pagedep_deps(struct vnode *, struct mount *,
	    struct diraddhd *);
STATIC	int flush_inodedep_deps(struct fs *, ufsino_t);
STATIC	int handle_written_filepage(struct pagedep *, struct buf *);
STATIC  void diradd_inode_written(struct diradd *, struct inodedep *);
STATIC	int handle_written_inodeblock(struct inodedep *, struct buf *);
STATIC	void handle_allocdirect_partdone(struct allocdirect *);
STATIC	void handle_allocindir_partdone(struct allocindir *);
STATIC	void initiate_write_filepage(struct pagedep *, struct buf *);
STATIC	void handle_written_mkdir(struct mkdir *, int);
STATIC	void initiate_write_inodeblock_ufs1(struct inodedep *, struct buf *);
#ifdef FFS2
STATIC	void initiate_write_inodeblock_ufs2(struct inodedep *, struct buf *);
#endif
STATIC	void handle_workitem_freefile(struct freefile *);
STATIC	void handle_workitem_remove(struct dirrem *);
STATIC	struct dirrem *newdirrem(struct buf *, struct inode *,
	    struct inode *, int, struct dirrem **);
STATIC	void free_diradd(struct diradd *);
STATIC	void free_allocindir(struct allocindir *, struct inodedep *);
STATIC	void free_newdirblk(struct newdirblk *);
STATIC	int indir_trunc(struct inode *, daddr_t, int, daddr_t, long *);
STATIC	void deallocate_dependencies(struct buf *, struct inodedep *);
STATIC	void free_allocdirect(struct allocdirectlst *,
	    struct allocdirect *, int);
STATIC	int check_inode_unwritten(struct inodedep *);
STATIC	int free_inodedep(struct inodedep *);
STATIC	void handle_workitem_freeblocks(struct freeblks *);
STATIC	void merge_inode_lists(struct inodedep *);
STATIC	void setup_allocindir_phase2(struct buf *, struct inode *,
	    struct allocindir *);
STATIC	struct allocindir *newallocindir(struct inode *, int, daddr_t,
	    daddr_t);
STATIC	void handle_workitem_freefrag(struct freefrag *);
STATIC	struct freefrag *newfreefrag(struct inode *, daddr_t, long);
STATIC	void allocdirect_merge(struct allocdirectlst *,
	    struct allocdirect *, struct allocdirect *);
STATIC	struct bmsafemap *bmsafemap_lookup(struct buf *);
STATIC	int newblk_lookup(struct fs *, daddr_t, int,
	    struct newblk **);
STATIC	int inodedep_lookup(struct fs *, ufsino_t, int, struct inodedep **);
STATIC	int pagedep_lookup(struct inode *, daddr_t, int, struct pagedep **);
STATIC	void pause_timer(void *);
STATIC	int request_cleanup(int, int);
STATIC	int process_worklist_item(struct mount *, int *, int);
STATIC	void add_to_worklist(struct worklist *);

/*
 * Exported softdep operations.
 */
void softdep_disk_io_initiation(struct buf *);
void softdep_disk_write_complete(struct buf *);
void softdep_deallocate_dependencies(struct buf *);
void softdep_move_dependencies(struct buf *, struct buf *);
int softdep_count_dependencies(struct buf *bp, int, int);

/*
 * Locking primitives.
 *
 * For a uniprocessor, all we need to do is protect against disk
 * interrupts. For a multiprocessor, this lock would have to be
 * a mutex. A single mutex is used throughout this file, though
 * finer grain locking could be used if contention warranted it.
 *
 * For a multiprocessor, the sleep call would accept a lock and
 * release it after the sleep processing was complete. In a uniprocessor
 * implementation there is no such interlock, so we simple mark
 * the places where it needs to be done with the `interlocked' form
 * of the lock calls. Since the uniprocessor sleep already interlocks
 * the spl, there is nothing that really needs to be done.
 */
#ifndef /* NOT */ DEBUG
STATIC struct lockit {
	int	lkt_spl;
} lk = { 0 };
#define ACQUIRE_LOCK(lk)		(lk)->lkt_spl = splbio()
#define FREE_LOCK(lk)			splx((lk)->lkt_spl)
#define ACQUIRE_LOCK_INTERLOCKED(lk,s)	(lk)->lkt_spl = (s)
#define FREE_LOCK_INTERLOCKED(lk) ((lk)->lkt_spl)

#else /* DEBUG */
STATIC struct lockit {
	int	lkt_spl;
	pid_t	lkt_held;
	int	lkt_line;
} lk = { 0, -1 };
STATIC int lockcnt;

STATIC	void acquire_lock(struct lockit *, int);
STATIC	void free_lock(struct lockit *, int);
STATIC	void acquire_lock_interlocked(struct lockit *, int, int);
STATIC	int free_lock_interlocked(struct lockit *, int);

#define ACQUIRE_LOCK(lk)		acquire_lock(lk, __LINE__)
#define FREE_LOCK(lk)			free_lock(lk, __LINE__)
#define ACQUIRE_LOCK_INTERLOCKED(lk,s)	acquire_lock_interlocked(lk, (s), __LINE__)
#define FREE_LOCK_INTERLOCKED(lk) free_lock_interlocked(lk, __LINE__)

STATIC void
acquire_lock(struct lockit *lk, int line)
{
	pid_t holder;
	int original_line;

	if (lk->lkt_held != -1) {
		holder = lk->lkt_held;
		original_line = lk->lkt_line;
		FREE_LOCK(lk);
		if (holder == CURPROC->p_tid)
			panic("softdep_lock: locking against myself, acquired at line %d, relocked at line %d", original_line, line);
		else
			panic("softdep_lock: lock held by %d, acquired at line %d, relocked at line %d", holder, original_line, line);
	}
	lk->lkt_spl = splbio();
	lk->lkt_held = CURPROC->p_tid;
	lk->lkt_line = line;
	lockcnt++;
}

STATIC void
free_lock(struct lockit *lk, int line)
{

	if (lk->lkt_held == -1)
		panic("softdep_unlock: lock not held at line %d", line);
	lk->lkt_held = -1;
	splx(lk->lkt_spl);
}

STATIC void
acquire_lock_interlocked(struct lockit *lk, int s, int line)
{
	pid_t holder;
	int original_line;

	if (lk->lkt_held != -1) {
		holder = lk->lkt_held;
		original_line = lk->lkt_line;
		FREE_LOCK_INTERLOCKED(lk);
		if (holder == CURPROC->p_tid)
			panic("softdep_lock: locking against myself, acquired at line %d, relocked at line %d", original_line, line);
		else
			panic("softdep_lock: lock held by %d, acquired at line %d, relocked at line %d", holder, original_line, line);
	}
	lk->lkt_held = CURPROC->p_tid;
	lk->lkt_line = line;
	lk->lkt_spl = s;
	lockcnt++;
}

STATIC int
free_lock_interlocked(struct lockit *lk, int line)
{

	if (lk->lkt_held == -1)
		panic("softdep_unlock_interlocked: lock not held at line %d", line);
	lk->lkt_held = -1;

	return (lk->lkt_spl);
}
#endif /* DEBUG */

/*
 * Place holder for real semaphores.
 */
struct sema {
	int	value;
	pid_t	holder;
	char	*name;
	int	prio;
};
STATIC	void sema_init(struct sema *, char *, int);
STATIC	int sema_get(struct sema *, struct lockit *);
STATIC	void sema_release(struct sema *);

STATIC void
sema_init(struct sema *semap, char *name, int prio)
{

	semap->holder = -1;
	semap->value = 0;
	semap->name = name;
	semap->prio = prio;
}

STATIC int
sema_get(struct sema *semap, struct lockit *interlock)
{
	int s;

	if (semap->value++ > 0) {
		if (interlock != NULL)
			s = FREE_LOCK_INTERLOCKED(interlock);
		tsleep_nsec(semap, semap->prio, semap->name, INFSLP);
		if (interlock != NULL) {
			ACQUIRE_LOCK_INTERLOCKED(interlock, s);
			FREE_LOCK(interlock);
		}
		return (0);
	}
	semap->holder = CURPROC->p_tid;
	if (interlock != NULL)
		FREE_LOCK(interlock);
	return (1);
}

STATIC void
sema_release(struct sema *semap)
{

	if (semap->value <= 0 || semap->holder != CURPROC->p_tid) {
#ifdef DEBUG
		if (lk.lkt_held != -1)
			FREE_LOCK(&lk);
#endif
		panic("sema_release: not held");
	}
	if (--semap->value > 0) {
		semap->value = 0;
		wakeup(semap);
	}
	semap->holder = -1;
}

/*
 * Memory management.
 */
STATIC struct pool pagedep_pool;
STATIC struct pool inodedep_pool;
STATIC struct pool newblk_pool;
STATIC struct pool bmsafemap_pool;
STATIC struct pool allocdirect_pool;
STATIC struct pool indirdep_pool;
STATIC struct pool allocindir_pool;
STATIC struct pool freefrag_pool;
STATIC struct pool freeblks_pool;
STATIC struct pool freefile_pool;
STATIC struct pool diradd_pool;
STATIC struct pool mkdir_pool;
STATIC struct pool dirrem_pool;
STATIC struct pool newdirblk_pool;

static __inline void
softdep_free(struct worklist *item, int type)
{

	switch (type) {
	case D_PAGEDEP:
		pool_put(&pagedep_pool, item);
		break;

	case D_INODEDEP:
		pool_put(&inodedep_pool, item);
		break;

	case D_BMSAFEMAP:
		pool_put(&bmsafemap_pool, item);
		break;

	case D_ALLOCDIRECT:
		pool_put(&allocdirect_pool, item);
		break;

	case D_INDIRDEP:
		pool_put(&indirdep_pool, item);
		break;

	case D_ALLOCINDIR:
		pool_put(&allocindir_pool, item);
		break;
	case D_FREEFRAG:
		pool_put(&freefrag_pool, item);
		break;

	case D_FREEBLKS:
		pool_put(&freeblks_pool, item);
		break;

	case D_FREEFILE:
		pool_put(&freefile_pool, item);
		break;

	case D_DIRADD:
		pool_put(&diradd_pool, item);
		break;

	case D_MKDIR:
		pool_put(&mkdir_pool, item);
		break;

	case D_DIRREM:
		pool_put(&dirrem_pool, item);
		break;

	case D_NEWDIRBLK:
		pool_put(&newdirblk_pool, item);
		break;

	default:
#ifdef DEBUG
		if (lk.lkt_held != -1)
			FREE_LOCK(&lk);
#endif
		panic("softdep_free: unknown type %d", type);
	}
}

struct workhead softdep_freequeue;

static __inline void
softdep_freequeue_add(struct worklist *item)
{
	int s;

	s = splbio();
	LIST_INSERT_HEAD(&softdep_freequeue, item, wk_list);
	splx(s);
}

static __inline void
softdep_freequeue_process(void)
{
	struct worklist *wk;

	splassert(IPL_BIO);

	while ((wk = LIST_FIRST(&softdep_freequeue)) != NULL) {
		LIST_REMOVE(wk, wk_list);
		FREE_LOCK(&lk);
		softdep_free(wk, wk->wk_type);
		ACQUIRE_LOCK(&lk);
	}
}

/*
 * Worklist queue management.
 * These routines require that the lock be held.
 */
#ifndef /* NOT */ DEBUG
#define WORKLIST_INSERT(head, item) do {	\
	(item)->wk_state |= ONWORKLIST;		\
	LIST_INSERT_HEAD(head, item, wk_list);	\
} while (0)
#define WORKLIST_REMOVE(item) do {		\
	(item)->wk_state &= ~ONWORKLIST;	\
	LIST_REMOVE(item, wk_list);		\
} while (0)
#define WORKITEM_FREE(item, type) softdep_freequeue_add((struct worklist *)item)

#else /* DEBUG */
STATIC	void worklist_insert(struct workhead *, struct worklist *);
STATIC	void worklist_remove(struct worklist *);
STATIC	void workitem_free(struct worklist *);

#define WORKLIST_INSERT(head, item) worklist_insert(head, item)
#define WORKLIST_REMOVE(item) worklist_remove(item)
#define WORKITEM_FREE(item, type) workitem_free((struct worklist *)item)

STATIC void
worklist_insert(struct workhead *head, struct worklist *item)
{

	if (lk.lkt_held == -1)
		panic("worklist_insert: lock not held");
	if (item->wk_state & ONWORKLIST) {
		FREE_LOCK(&lk);
		panic("worklist_insert: already on list");
	}
	item->wk_state |= ONWORKLIST;
	LIST_INSERT_HEAD(head, item, wk_list);
}

STATIC void
worklist_remove(struct worklist *item)
{

	if (lk.lkt_held == -1)
		panic("worklist_remove: lock not held");
	if ((item->wk_state & ONWORKLIST) == 0) {
		FREE_LOCK(&lk);
		panic("worklist_remove: not on list");
	}
	item->wk_state &= ~ONWORKLIST;
	LIST_REMOVE(item, wk_list);
}

STATIC void
workitem_free(struct worklist *item)
{

	if (item->wk_state & ONWORKLIST) {
		if (lk.lkt_held != -1)
			FREE_LOCK(&lk);
		panic("workitem_free: still on list");
	}
	softdep_freequeue_add(item);
}
#endif /* DEBUG */

/*
 * Workitem queue management
 */
STATIC struct workhead softdep_workitem_pending;
STATIC struct worklist *worklist_tail;
STATIC int num_on_worklist;	/* number of worklist items to be processed */
STATIC int softdep_worklist_busy; /* 1 => trying to do unmount */
STATIC int softdep_worklist_req; /* serialized waiters */
STATIC int max_softdeps;  /* maximum number of structs before slowdown */
STATIC int tickdelay = 2; /* number of ticks to pause during slowdown */
STATIC int proc_waiting;  /* tracks whether we have a timeout posted */
STATIC int *stat_countp;  /* statistic to count in proc_waiting timeout */
STATIC struct timeout proc_waiting_timeout;
STATIC struct proc *filesys_syncer; /* proc of filesystem syncer process */
STATIC int req_clear_inodedeps;	/* syncer process flush some inodedeps */
#define FLUSH_INODES	1
STATIC int req_clear_remove;	/* syncer process flush some freeblks */
#define FLUSH_REMOVE	2
/*
 * runtime statistics
 */
STATIC int stat_worklist_push;	/* number of worklist cleanups */
STATIC int stat_blk_limit_push;	/* number of times block limit neared */
STATIC int stat_ino_limit_push;	/* number of times inode limit neared */
STATIC int stat_blk_limit_hit;	/* number of times block slowdown imposed */
STATIC int stat_ino_limit_hit;	/* number of times inode slowdown imposed */
STATIC int stat_sync_limit_hit;	/* number of synchronous slowdowns imposed */
STATIC int stat_indir_blk_ptrs;	/* bufs redirtied as indir ptrs not written */
STATIC int stat_inode_bitmap;	/* bufs redirtied as inode bitmap not written */
STATIC int stat_direct_blk_ptrs;/* bufs redirtied as direct ptrs not written */
STATIC int stat_dir_entry;	/* bufs redirtied as dir entry cannot write */

/*
 * Add an item to the end of the work queue.
 * This routine requires that the lock be held.
 * This is the only routine that adds items to the list.
 * The following routine is the only one that removes items
 * and does so in order from first to last.
 */
STATIC void
add_to_worklist(struct worklist *wk)
{

	if (wk->wk_state & ONWORKLIST) {
#ifdef DEBUG
		if (lk.lkt_held != -1)
			FREE_LOCK(&lk);
#endif
		panic("add_to_worklist: already on list");
	}
	wk->wk_state |= ONWORKLIST;
	if (LIST_FIRST(&softdep_workitem_pending) == NULL)
		LIST_INSERT_HEAD(&softdep_workitem_pending, wk, wk_list);
	else
		LIST_INSERT_AFTER(worklist_tail, wk, wk_list);
	worklist_tail = wk;
	num_on_worklist += 1;
}

/*
 * Process that runs once per second to handle items in the background queue.
 *
 * Note that we ensure that everything is done in the order in which they
 * appear in the queue. The code below depends on this property to ensure
 * that blocks of a file are freed before the inode itself is freed. This
 * ordering ensures that no new <vfsid, inum, lbn> triples will be generated
 * until all the old ones have been purged from the dependency lists.
 */
int
softdep_process_worklist(struct mount *matchmnt)
{
	struct proc *p = CURPROC;
	int matchcnt, loopcount;
	struct timeval starttime;

	/*
	 * First process any items on the delayed-free queue.
	 */
	ACQUIRE_LOCK(&lk);
	softdep_freequeue_process();
	FREE_LOCK(&lk);

	/*
	 * Record the process identifier of our caller so that we can give
	 * this process preferential treatment in request_cleanup below.
	 * We can't do this in softdep_initialize, because the syncer doesn't
	 * have to run then.
	 * NOTE! This function _could_ be called with a curproc != syncerproc.
	 */
	filesys_syncer = syncerproc;
	matchcnt = 0;

	/*
	 * There is no danger of having multiple processes run this
	 * code, but we have to single-thread it when softdep_flushfiles()
	 * is in operation to get an accurate count of the number of items
	 * related to its mount point that are in the list.
	 */
	if (matchmnt == NULL) {
		if (softdep_worklist_busy < 0)
			return(-1);
		softdep_worklist_busy += 1;
	}

	/*
	 * If requested, try removing inode or removal dependencies.
	 */
	if (req_clear_inodedeps) {
		clear_inodedeps(p);
		req_clear_inodedeps -= 1;
		wakeup_one(&proc_waiting);
	}
	if (req_clear_remove) {
		clear_remove(p);
		req_clear_remove -= 1;
		wakeup_one(&proc_waiting);
	}
	loopcount = 1;
	getmicrouptime(&starttime);
	while (num_on_worklist > 0) {
		if (process_worklist_item(matchmnt, &matchcnt, LK_NOWAIT) == 0)
			break;

		/*
		 * If a umount operation wants to run the worklist
		 * accurately, abort.
		 */
		if (softdep_worklist_req && matchmnt == NULL) {
			matchcnt = -1;
			break;
		}

		/*
		 * If requested, try removing inode or removal dependencies.
		 */
		if (req_clear_inodedeps) {
			clear_inodedeps(p);
			req_clear_inodedeps -= 1;
			wakeup_one(&proc_waiting);
		}
		if (req_clear_remove) {
			clear_remove(p);
			req_clear_remove -= 1;
			wakeup_one(&proc_waiting);
		}
		/*
		 * We do not generally want to stop for buffer space, but if
		 * we are really being a buffer hog, we will stop and wait.
		 */
#if 0
		if (loopcount++ % 128 == 0)
			bwillwrite();
#endif
		/*
		 * Never allow processing to run for more than one
		 * second. Otherwise the other syncer tasks may get
		 * excessively backlogged.
		 */
		{
			struct timeval diff;
			struct timeval tv;

			getmicrouptime(&tv);
			timersub(&tv, &starttime, &diff);
			if (diff.tv_sec != 0 && matchmnt == NULL) {
				matchcnt = -1;
				break;
			}
		}

		/*
		 * Process any new items on the delayed-free queue.
		 */
		ACQUIRE_LOCK(&lk);
		softdep_freequeue_process();
```

```c
			FREE_LOCK(&lk);
	}
	if (matchmnt == NULL) {
		softdep_worklist_busy -= 1;
		if (softdep_worklist_req && softdep_worklist_busy == 0)
			wakeup(&softdep_worklist_req);
	}
	return (matchcnt);
}

/*
 * Process one item on the worklist.
 */
STATIC int
process_worklist_item(struct mount *matchmnt, int *matchcnt, int flags)
{
	struct worklist *wk, *wkend;
	struct dirrem *dirrem;
	struct mount *mp;
	struct vnode *vp;

	ACQUIRE_LOCK(&lk);
	/*
	 * Normally we just process each item on the worklist in order.
	 * However, if we are in a situation where we cannot lock any
	 * inodes, we have to skip over any dirrem requests whose
	 * vnodes are resident and locked.
	 */
	LIST_FOREACH(wk, &softdep_workitem_pending, wk_list) {
		if ((flags & LK_NOWAIT) == 0 || wk->wk_type != D_DIRREM)
			break;
		dirrem = WK_DIRREM(wk);
		vp = ufs_ihashlookup(VFSTOUFS(dirrem->dm_mnt)->um_dev,
		    dirrem->dm_oldinum);
		if (vp == NULL || !VOP_ISLOCKED(vp))
			break;
	}
	if (wk == NULL) {
		FREE_LOCK(&lk);
		return (0);
	}
	/*
	 * Remove the item to be processed. If we are removing the last
	 * item on the list, we need to recalculate the tail pointer.
	 * As this happens rarely and usually when the list is short,
	 * we just run down the list to find it rather than tracking it
	 * in the above loop.
	 */
	WORKLIST_REMOVE(wk);
	if (wk == worklist_tail) {
		LIST_FOREACH(wkend, &softdep_workitem_pending, wk_list)
			if (LIST_NEXT(wkend, wk_list) == NULL)
				break;
		worklist_tail = wkend;
	}
	num_on_worklist -= 1;
	FREE_LOCK(&lk);
	switch (wk->wk_type) {

	case D_DIRREM:
		/* removal of a directory entry */
		mp = WK_DIRREM(wk)->dm_mnt;
#if 0
		if (vn_write_suspend_wait(NULL, mp, V_NOWAIT))
			panic("%s: dirrem on suspended filesystem",
				"process_worklist_item");
#endif
		if (matchmnt != NULL && mp == matchmnt)
			*matchcnt += 1;
		handle_workitem_remove(WK_DIRREM(wk));
		break;

	case D_FREEBLKS:
		/* releasing blocks and/or fragments from a file */
		mp = WK_FREEBLKS(wk)->fb_mnt;
#if 0
		if (vn_write_suspend_wait(NULL, mp, V_NOWAIT))
			panic("%s: freeblks on suspended filesystem",
				"process_worklist_item");
#endif
		if (matchmnt != NULL && mp == matchmnt)
			*matchcnt += 1;
		handle_workitem_freeblocks(WK_FREEBLKS(wk));
		break;

	case D_FREEFRAG:
		/* releasing a fragment when replaced as a file grows */
		mp = WK_FREEFRAG(wk)->ff_mnt;
#if 0
		if (vn_write_suspend_wait(NULL, mp, V_NOWAIT))
			panic("%s: freefrag on suspended filesystem",
				"process_worklist_item");
#endif
		if (matchmnt != NULL && mp == matchmnt)
			*matchcnt += 1;
		handle_workitem_freefrag(WK_FREEFRAG(wk));
		break;

	case D_FREEFILE:
		/* releasing an inode when its link count drops to 0 */
		mp = WK_FREEFILE(wk)->fx_mnt;
#if 0
		if (vn_write_suspend_wait(NULL, mp, V_NOWAIT))
			panic("%s: freefile on suspended filesystem",
				"process_worklist_item");
#endif
		if (matchmnt != NULL && mp == matchmnt)
			*matchcnt += 1;
		handle_workitem_freefile(WK_FREEFILE(wk));
		break;

	default:
		panic("%s_process_worklist: Unknown type %s",
		    "softdep", TYPENAME(wk->wk_type));
		/* NOTREACHED */
	}
	return (1);
}

/*
 * Move dependencies from one buffer to another.
 */
void
softdep_move_dependencies(struct buf *oldbp, struct buf *newbp)
{
	struct worklist *wk, *wktail;

	if (LIST_FIRST(&newbp->b_dep) != NULL)
		panic("softdep_move_dependencies: need merge code");
	wktail = NULL;
	ACQUIRE_LOCK(&lk);
	while ((wk = LIST_FIRST(&oldbp->b_dep)) != NULL) {
		LIST_REMOVE(wk, wk_list);
		if (wktail == NULL)
			LIST_INSERT_HEAD(&newbp->b_dep, wk, wk_list);
		else
			LIST_INSERT_AFTER(wktail, wk, wk_list);
		wktail = wk;
	}
	FREE_LOCK(&lk);
}

/*
 * Purge the work list of all items associated with a particular mount point.
 */
int
softdep_flushworklist(struct mount *oldmnt, int *countp, struct proc *p)
{
	struct vnode *devvp;
	int count, error = 0;

	/*
	 * Await our turn to clear out the queue, then serialize access.
	 */
	while (softdep_worklist_busy) {
		softdep_worklist_req += 1;
		tsleep_nsec(&softdep_worklist_req, PRIBIO, "softflush", INFSLP);
		softdep_worklist_req -= 1;
	}
	softdep_worklist_busy = -1;
	/*
	 * Alternately flush the block device associated with the mount
	 * point and process any dependencies that the flushing
	 * creates. We continue until no more worklist dependencies
	 * are found.
	 */
	*countp = 0;
	devvp = VFSTOUFS(oldmnt)->um_devvp;
	while ((count = softdep_process_worklist(oldmnt)) > 0) {
		*countp += count;
		vn_lock(devvp, LK_EXCLUSIVE | LK_RETRY);
		error = VOP_FSYNC(devvp, p->p_ucred, MNT_WAIT, p);
		VOP_UNLOCK(devvp);
		if (error)
			break;
	}
	softdep_worklist_busy = 0;
	if (softdep_worklist_req)
		wakeup(&softdep_worklist_req);
	return (error);
}

/*
 * Flush all vnodes and worklist items associated with a specified mount point.
 */
int
softdep_flushfiles(struct mount *oldmnt, int flags, struct proc *p)
{
	int error, count, loopcnt;

	/*
	 * Alternately flush the vnodes associated with the mount
	 * point and process any dependencies that the flushing
	 * creates. In theory, this loop can happen at most twice,
	 * but we give it a few extra just to be sure.
	 */
	for (loopcnt = 10; loopcnt > 0; loopcnt--) {
		/*
		 * Do another flush in case any vnodes were brought in
		 * as part of the cleanup operations.
		 */
		if ((error = ffs_flushfiles(oldmnt, flags, p)) != 0)
			break;
		if ((error = softdep_flushworklist(oldmnt, &count, p)) != 0 ||
		    count == 0)
			break;
	}
	/*
	 * If the reboot process sleeps during the loop, the update
	 * process may call softdep_process_worklist() and create
	 * new dirty vnodes at the mount point.  Call ffs_flushfiles()
	 * again after the loop has flushed all soft dependencies.
	 */
	if (error == 0)
		error = ffs_flushfiles(oldmnt, flags, p);
	/*
	 * If we are unmounting then it is an error to fail. If we
	 * are simply trying to downgrade to read-only, then filesystem
	 * activity can keep us busy forever, so we just fail with EBUSY.
	 */
	if (loopcnt == 0) {
		error = EBUSY;
	}
	return (error);
}

/*
 * Structure hashing.
 *
 * There are three types of structures that can be looked up:
 *	1) pagedep structures identified by mount point, inode number,
 *	   and logical block.
 *	2) inodedep structures identified by mount point and inode number.
 *	3) newblk structures identified by mount point and
 *	   physical block number.
 *
 * The "pagedep" and "inodedep" dependency structures are hashed
 * separately from the file blocks and inodes to which they correspond.
 * This separation helps when the in-memory copy of an inode or
 * file block must be replaced. It also obviates the need to access
 * an inode or file page when simply updating (or de-allocating)
 * dependency structures. Lookup of newblk structures is needed to
 * find newly allocated blocks when trying to associate them with
 * their allocdirect or allocindir structure.
 *
 * The lookup routines optionally create and hash a new instance when
 * an existing entry is not found.
 */
#define DEPALLOC 0x0001	/* allocate structure if lookup fails */
#define NODELAY		0x0002	/* cannot do background work */

SIPHASH_KEY softdep_hashkey;

/*
 * Structures and routines associated with pagedep caching.
 */
LIST_HEAD(pagedep_hashhead, pagedep) *pagedep_hashtbl;
u_long	pagedep_hash;		/* size of hash table - 1 */
STATIC struct sema pagedep_in_progress;

/*
 * Look up a pagedep. Return 1 if found, 0 if not found or found
 * when asked to allocate but not associated with any buffer.
 * If not found, allocate if DEPALLOC flag is passed.
 * Found or allocated entry is returned in pagedeppp.
 * This routine must be called with splbio interrupts blocked.
 */
STATIC int
pagedep_lookup(struct inode *ip, daddr_t lbn, int flags,
    struct pagedep **pagedeppp)
{
	SIPHASH_CTX ctx;
	struct pagedep *pagedep;
	struct pagedep_hashhead *pagedephd;
	struct mount *mp;
	int i;

	splassert(IPL_BIO);

#ifdef DEBUG
	if (lk.lkt_held == -1)
		panic("pagedep_lookup: lock not held");
#endif
	mp = ITOV(ip)->v_mount;

	SipHash24_Init(&ctx, &softdep_hashkey);
	SipHash24_Update(&ctx, &mp, sizeof(mp));
	SipHash24_Update(&ctx, &ip->i_number, sizeof(ip->i_number));
	SipHash24_Update(&ctx, &lbn, sizeof(lbn));
	pagedephd = &pagedep_hashtbl[SipHash24_End(&ctx) & pagedep_hash];
top:
	LIST_FOREACH(pagedep, pagedephd, pd_hash)
		if (ip->i_number == pagedep->pd_ino &&
		    lbn == pagedep->pd_lbn &&
		    mp == pagedep->pd_mnt)
			break;
	if (pagedep) {
		*pagedeppp = pagedep;
		if ((flags & DEPALLOC) != 0 &&
		    (pagedep->pd_state & ONWORKLIST) == 0)
			return (0);
		return (1);
	}
	if ((flags & DEPALLOC) == 0) {
```

```c
                *pagedeppp = NULL;
                return (0);
        }
        if (sema_get(&pagedep_in_progress, &lk) == 0) {
                ACQUIRE_LOCK(&lk);
                goto top;
        }
        pagedep = pool_get(&pagedep_pool, PR_WAITOK | PR_ZERO);
        pagedep->pd_list.wk_type = D_PAGEDEP;
        pagedep->pd_mnt = mp;
        pagedep->pd_ino = ip->i_number;
        pagedep->pd_lbn = lbn;
        LIST_INIT(&pagedep->pd_dirremhd);
        LIST_INIT(&pagedep->pd_pendinghd);
        for (i = 0; i < DAHASHSZ; i++)
                LIST_INIT(&pagedep->pd_diraddhd[i]);
        ACQUIRE_LOCK(&lk);
        LIST_INSERT_HEAD(pagedephd, pagedep, pd_hash);
        sema_release(&pagedep_in_progress);
        *pagedeppp = pagedep;
        return (0);
}

/*
 * Structures and routines associated with inodedep caching.
 */
LIST_HEAD(inodedep_hashhead, inodedep) *inodedep_hashtbl;
STATIC u_long   inodedep_hash;  /* size of hash table - 1 */
STATIC long     num_inodedep;   /* number of inodedep allocated */
STATIC struct sema inodedep_in_progress;

/*
 * Look up a inodedep. Return 1 if found, 0 if not found.
 * If not found, allocate if DEPALLOC flag is passed.
 * Found or allocated entry is returned in inodedeppp.
 * This routine must be called with splbio interrupts blocked.
 */
STATIC int
inodedep_lookup(struct fs *fs, ufsino_t inum, int flags,
    struct inodedep **inodedeppp)
{
        SIPHASH_CTX ctx;
        struct inodedep *inodedep;
        struct inodedep_hashhead *inodedephd;
        int firsttry;

        splassert(IPL_BIO);

#ifdef DEBUG
        if (lk.lkt_held == -1)
                panic("inodedep_lookup: lock not held");
#endif
        firsttry = 1;
        SipHash24_Init(&ctx, &softdep_hashkey);
        SipHash24_Update(&ctx, &fs, sizeof(fs));
        SipHash24_Update(&ctx, &inum, sizeof(inum));
        inodedephd = &inodedep_hashtbl[SipHash24_End(&ctx) & inodedep_hash];
top:
        LIST_FOREACH(inodedep, inodedephd, id_hash)
                if (inum == inodedep->id_ino && fs == inodedep->id_fs)
                        break;
        if (inodedep) {
                *inodedeppp = inodedep;
                return (1);
        }
        if ((flags & DEPALLOC) == 0) {
                *inodedeppp = NULL;
                return (0);
        }
        /*
         * If we are over our limit, try to improve the situation.
         */
        if (num_inodedep > max_softdeps && firsttry && (flags & NODELAY) == 0 &&
            request_cleanup(FLUSH_INODES, 1)) {
                firsttry = 0;
                goto top;
        }
        if (sema_get(&inodedep_in_progress, &lk) == 0) {
                ACQUIRE_LOCK(&lk);
                goto top;
        }
        num_inodedep += 1;
        inodedep = pool_get(&inodedep_pool, PR_WAITOK);
        inodedep->id_list.wk_type = D_INODEDEP;
        inodedep->id_fs = fs;
        inodedep->id_ino = inum;
        inodedep->id_state = ALLCOMPLETE;
        inodedep->id_nlinkdelta = 0;
        inodedep->id_savedino1 = NULL;
        inodedep->id_savedsize = -1;
        inodedep->id_buf = NULL;
        LIST_INIT(&inodedep->id_pendinghd);
        LIST_INIT(&inodedep->id_inowait);
        LIST_INIT(&inodedep->id_bufwait);
        TAILQ_INIT(&inodedep->id_inoupdt);
        TAILQ_INIT(&inodedep->id_newinoupdt);
        ACQUIRE_LOCK(&lk);
        LIST_INSERT_HEAD(inodedephd, inodedep, id_hash);
        sema_release(&inodedep_in_progress);
        *inodedeppp = inodedep;
        return (0);
}

/*
 * Structures and routines associated with newblk caching.
 */
LIST_HEAD(newblk_hashhead, newblk) *newblk_hashtbl;
u_long  newblk_hash;            /* size of hash table - 1 */
STATIC struct sema newblk_in_progress;

/*
 * Look up a newblk. Return 1 if found, 0 if not found.
 * If not found, allocate if DEPALLOC flag is passed.
 * Found or allocated entry is returned in newblkpp.
 */
STATIC int
newblk_lookup(struct fs *fs, daddr_t newblkno, int flags,
    struct newblk **newblkpp)
{
        SIPHASH_CTX ctx;
        struct newblk *newblk;
        struct newblk_hashhead *newblkhd;

        SipHash24_Init(&ctx, &softdep_hashkey);
        SipHash24_Update(&ctx, &fs, sizeof(fs));
        SipHash24_Update(&ctx, &newblkno, sizeof(newblkno));
        newblkhd = &newblk_hashtbl[SipHash24_End(&ctx) & newblk_hash];
top:
        LIST_FOREACH(newblk, newblkhd, nb_hash)
                if (newblkno == newblk->nb_newblkno && fs == newblk->nb_fs)
                        break;
        if (newblk) {
                *newblkpp = newblk;
                return (1);
        }
        if ((flags & DEPALLOC) == 0) {
                *newblkpp = NULL;
                return (0);
        }
        if (sema_get(&newblk_in_progress, NULL) == 0)
                goto top;
        newblk = pool_get(&newblk_pool, PR_WAITOK);
        newblk->nb_state = 0;
        newblk->nb_fs = fs;
        newblk->nb_newblkno = newblkno;
        LIST_INSERT_HEAD(newblkhd, newblk, nb_hash);
        sema_release(&newblk_in_progress);
        *newblkpp = newblk;
        return (0);
}

/*
 * Executed during filesystem system initialization before
 * mounting any file systems.
 */
void
softdep_initialize(void)
{

        bioops.io_start = softdep_disk_io_initiation;
        bioops.io_complete = softdep_disk_write_complete;
        bioops.io_deallocate = softdep_deallocate_dependencies;
        bioops.io_movedeps = softdep_move_dependencies;
        bioops.io_countdeps = softdep_count_dependencies;

        LIST_INIT(&mkdirlisthd);
        LIST_INIT(&softdep_workitem_pending);
#ifdef KMEMSTATS
        max_softdeps = min (initialvnodes * 8,
            kmemstats[M_INODEDEP].ks_limit / (2 * sizeof(struct inodedep)));
#else
        max_softdeps = initialvnodes * 4;
#endif
        arc4random_buf(&softdep_hashkey, sizeof(softdep_hashkey));
        pagedep_hashtbl = hashinit(initialvnodes / 5, M_PAGEDEP, M_WAITOK,
            &pagedep_hash);
        sema_init(&pagedep_in_progress, "pagedep", PRIBIO);
        inodedep_hashtbl = hashinit(initialvnodes, M_INODEDEP, M_WAITOK,
            &inodedep_hash);
        sema_init(&inodedep_in_progress, "inodedep", PRIBIO);
        newblk_hashtbl = hashinit(64, M_NEWBLK, M_WAITOK, &newblk_hash);
        sema_init(&newblk_in_progress, "newblk", PRIBIO);
        timeout_set(&proc_waiting_timeout, pause_timer, NULL);
        pool_init(&pagedep_pool, sizeof(struct pagedep), 0, IPL_NONE,
            PR_WAITOK, "pagedep", NULL);
        pool_init(&inodedep_pool, sizeof(struct inodedep), 0, IPL_NONE,
            PR_WAITOK, "inodedep", NULL);
        pool_init(&newblk_pool, sizeof(struct newblk), 0, IPL_NONE,
            PR_WAITOK, "newblk", NULL);
        pool_init(&bmsafemap_pool, sizeof(struct bmsafemap), 0, IPL_NONE,
            PR_WAITOK, "bmsafemap", NULL);
        pool_init(&allocdirect_pool, sizeof(struct allocdirect), 0, IPL_NONE,
            PR_WAITOK, "allocdir", NULL);
        pool_init(&indirdep_pool, sizeof(struct indirdep), 0, IPL_NONE,
            PR_WAITOK, "indirdep", NULL);
        pool_init(&allocindir_pool, sizeof(struct allocindir), 0, IPL_NONE,
            PR_WAITOK, "allocindir", NULL);
        pool_init(&freefrag_pool, sizeof(struct freefrag), 0, IPL_NONE,
            PR_WAITOK, "freefrag", NULL);
        pool_init(&freeblks_pool, sizeof(struct freeblks), 0, IPL_NONE,
            PR_WAITOK, "freeblks", NULL);
        pool_init(&freefile_pool, sizeof(struct freefile), 0, IPL_NONE,
            PR_WAITOK, "freefile", NULL);
        pool_init(&diradd_pool, sizeof(struct diradd), 0, IPL_NONE,
            PR_WAITOK, "diradd", NULL);
        pool_init(&mkdir_pool, sizeof(struct mkdir), 0, IPL_NONE,
            PR_WAITOK, "mkdir", NULL);
        pool_init(&dirrem_pool, sizeof(struct dirrem), 0, IPL_NONE,
            PR_WAITOK, "dirrem", NULL);
        pool_init(&newdirblk_pool, sizeof(struct newdirblk), 0, IPL_NONE,
            PR_WAITOK, "newdirblk", NULL);
}

/*
 * Called at mount time to notify the dependency code that a
 * filesystem wishes to use it.
 */
int
softdep_mount(struct vnode *devvp, struct mount *mp, struct fs *fs,
    struct ucred *cred)
{
        struct csum_total cstotal;
        struct cg *cgp;
        struct buf *bp;
        int error, cyl;

        /*
         * When doing soft updates, the counters in the
         * superblock may have gotten out of sync, so we have
         * to scan the cylinder groups and recalculate them.
         */
        if ((fs->fs_flags & FS_UNCLEAN) == 0)
                return (0);
        memset(&cstotal, 0, sizeof(cstotal));
        for (cyl = 0; cyl < fs->fs_ncg; cyl++) {
                if ((error = bread(devvp, fsbtodb(fs, cgtod(fs, cyl)),
                    fs->fs_cgsize, &bp)) != 0) {
                        brelse(bp);
                        return (error);
                }
                cgp = (struct cg *)bp->b_data;
                cstotal.cs_nffree += cgp->cg_cs.cs_nffree;
                cstotal.cs_nbfree += cgp->cg_cs.cs_nbfree;
                cstotal.cs_nifree += cgp->cg_cs.cs_nifree;
                cstotal.cs_ndir += cgp->cg_cs.cs_ndir;
                fs->fs_cs(fs, cyl) = cgp->cg_cs;
                brelse(bp);
        }
#ifdef DEBUG
        if (memcmp(&cstotal, &fs->fs_cstotal, sizeof(cstotal)))
                printf("ffs_mountfs: superblock updated for soft updates\n");
#endif
        memcpy(&fs->fs_cstotal, &cstotal, sizeof(cstotal));
        return (0);
}

/*
 * Protecting the freemaps (or bitmaps).
 *
 * To eliminate the need to execute fsck before mounting a file system
 * after a power failure, one must (conservatively) guarantee that the
 * on-disk copy of the bitmaps never indicate that a live inode or block is
 * free.  So, when a block or inode is allocated, the bitmap should be
 * updated (on disk) before any new pointers.  When a block or inode is
 * freed, the bitmap should not be updated until all pointers have been
 * reset.  The latter dependency is handled by the delayed de-allocation
 * approach described below for block and inode de-allocation.  The former
 * dependency is handled by calling the following procedure when a block or
 * inode is allocated. When an inode is allocated an "inodedep" is created
 * with its DEPCOMPLETE flag cleared until its bitmap is written to disk.
 * Each "inodedep" is also inserted into the hash indexing structure so
 * that any additional link additions can be made dependent on the inode
 * allocation.
 *
 * The ufs file system maintains a number of free block counts (e.g., per
 * cylinder group, per cylinder and per <cylinder, rotational position> pair)
 * in addition to the bitmaps.  These counts are used to improve efficiency
 * during allocation and therefore must be consistent with the bitmaps.
 * There is no convenient way to guarantee post-crash consistency of these
 * counts with simple update ordering, for two main reasons: (1) The counts
 * and bitmaps for a single cylinder group block are not in the same disk
 * sector.  If a disk write is interrupted (e.g., by power failure), one may
 * be written and the other not.  (2) Some of the counts are located in the
 * superblock rather than the cylinder group block. So, we focus our soft
 * updates implementation on protecting the bitmaps. When mounting a
 * filesystem, we recompute the auxiliary counts from the bitmaps.
 */

/*
 * Called just after updating the cylinder group block to allocate an inode.
 */
/* buffer for cylgroup block with inode map */
/* inode related to allocation */
/* new inode number being allocated */
void
softdep_setup_inomapdep(struct buf *bp, struct inode *ip, ufsino_t newinum)
{
        struct inodedep *inodedep;
        struct bmsafemap *bmsafemap;

        /*
         * Create a dependency for the newly allocated inode.
         * Panic if it already exists as something is seriously wrong.
         * Otherwise add it to the dependency list for the buffer holding
```

```c
	 * the cylinder group map from which it was allocated.
	 */
	ACQUIRE_LOCK(&lk);
	if (inodedep_lookup(ip->i_fs, newinum, DEPALLOC | NODELAY, &inodedep)
	    != 0) {
		FREE_LOCK(&lk);
		panic("softdep_setup_inomapdep: found inode");
	}
	inodedep->id_buf = bp;
	inodedep->id_state &= ~DEPCOMPLETE;
	bmsafemap = bmsafemap_lookup(bp);
	LIST_INSERT_HEAD(&bmsafemap->sm_inodedephd, inodedep, id_deps);
	FREE_LOCK(&lk);
}

/*
 * Called just after updating the cylinder group block to
 * allocate block or fragment.
 */
/* buffer for cylgroup block with block map */
/* filesystem doing allocation */
/* number of newly allocated block */
void
softdep_setup_blkmapdep(struct buf *bp, struct fs *fs, daddr_t newblkno)
{
	struct newblk *newblk;
	struct bmsafemap *bmsafemap;

	/*
	 * Create a dependency for the newly allocated block.
	 * Add it to the dependency list for the buffer holding
	 * the cylinder group map from which it was allocated.
	 */
	if (newblk_lookup(fs, newblkno, DEPALLOC, &newblk) != 0)
		panic("softdep_setup_blkmapdep: found block");
	ACQUIRE_LOCK(&lk);
	newblk->nb_bmsafemap = bmsafemap = bmsafemap_lookup(bp);
	LIST_INSERT_HEAD(&bmsafemap->sm_newblkhd, newblk, nb_deps);
	FREE_LOCK(&lk);
}

/*
 * Find the bmsafemap associated with a cylinder group buffer.
 * If none exists, create one. The buffer must be locked when
 * this routine is called and this routine must be called with
 * splbio interrupts blocked.
 */
STATIC struct bmsafemap *
bmsafemap_lookup(struct buf *bp)
{
	struct bmsafemap *bmsafemap;
	struct worklist *wk;

	splassert(IPL_BIO);

#ifdef DEBUG
	if (lk.lkt_held == -1)
		panic("bmsafemap_lookup: lock not held");
#endif
	LIST_FOREACH(wk, &bp->b_dep, wk_list)
		if (wk->wk_type == D_BMSAFEMAP)
			return (WK_BMSAFEMAP(wk));
	FREE_LOCK(&lk);
	bmsafemap = pool_get(&bmsafemap_pool, PR_WAITOK);
	bmsafemap->sm_list.wk_type = D_BMSAFEMAP;
	bmsafemap->sm_list.wk_state = 0;
	bmsafemap->sm_buf = bp;
	LIST_INIT(&bmsafemap->sm_allocdirecthd);
	LIST_INIT(&bmsafemap->sm_allocindirhd);
	LIST_INIT(&bmsafemap->sm_inodedephd);
	LIST_INIT(&bmsafemap->sm_newblkhd);
	ACQUIRE_LOCK(&lk);
	WORKLIST_INSERT(&bp->b_dep, &bmsafemap->sm_list);
	return (bmsafemap);
}

/*
 * Direct block allocation dependencies.
 *
 * When a new block is allocated, the corresponding disk locations must be
 * initialized (with zeros or new data) before the on-disk inode points to
 * them.  Also, the freemap from which the block was allocated must be
 * updated (on disk) before the inode's pointer. These two dependencies are
 * independent of each other and are needed for all file blocks and indirect
 * blocks that are pointed to directly by the inode.  Just before the
 * "in-core" version of the inode is updated with a newly allocated block
 * number, a procedure (below) is called to setup allocation dependency
 * structures.  These structures are removed when the corresponding
 * dependencies are satisfied or when the block allocation becomes obsolete
 * (i.e., the file is deleted, the block is de-allocated, or the block is a
 * fragment that gets upgraded).  All of these cases are handled in
 * procedures described later.
 *
 * When a file extension causes a fragment to be upgraded, either to a larger
 * fragment or to a full block, the on-disk location may change (if the
 * previous fragment could not simply be extended). In this case, the old
 * fragment must be de-allocated, but not until after the inode's pointer has
 * been updated. In most cases, this is handled by later procedures, which
 * will construct a "freefrag" structure to be added to the workitem queue
 * when the inode update is complete (or obsolete).  The main exception to
 * this is when an allocation occurs while a pending allocation dependency
 * (for the same block pointer) remains.  This case is handled in the main
 * allocation dependency setup procedure by immediately freeing the
 * unreferenced fragments.
 */
/* inode to which block is being added */
/* block pointer within inode */
/* disk block number being added */
/* previous block number, 0 unless frag */
/* size of new block */
/* size of new block */
/* bp for allocated block */
void
softdep_setup_allocdirect(struct inode *ip, daddr_t lbn, daddr_t newblkno,
    daddr_t oldblkno, long newsize, long oldsize, struct buf *bp)
{
	struct allocdirect *adp, *oldadp;
	struct allocdirectlst *adphead;
	struct bmsafemap *bmsafemap;
	struct inodedep *inodedep;
	struct pagedep *pagedep;
	struct newblk *newblk;

	adp = pool_get(&allocdirect_pool, PR_WAITOK | PR_ZERO);
	adp->ad_list.wk_type = D_ALLOCDIRECT;
	adp->ad_lbn = lbn;
	adp->ad_newblkno = newblkno;
	adp->ad_oldblkno = oldblkno;
	adp->ad_newsize = newsize;
	adp->ad_oldsize = oldsize;
	adp->ad_state = ATTACHED;
	LIST_INIT(&adp->ad_newdirblk);
	if (newblkno == oldblkno)
		adp->ad_freefrag = NULL;
	else
		adp->ad_freefrag = newfreefrag(ip, oldblkno, oldsize);

	if (newblk_lookup(ip->i_fs, newblkno, 0, &newblk) == 0)
		panic("softdep_setup_allocdirect: lost block");

	ACQUIRE_LOCK(&lk);
	inodedep_lookup(ip->i_fs, ip->i_number, DEPALLOC | NODELAY, &inodedep);
	adp->ad_inodedep = inodedep;

	if (newblk->nb_state == DEPCOMPLETE) {
		adp->ad_state |= DEPCOMPLETE;
		adp->ad_buf = NULL;
	} else {
		bmsafemap = newblk->nb_bmsafemap;
		adp->ad_buf = bmsafemap->sm_buf;
		LIST_REMOVE(newblk, nb_deps);
		LIST_INSERT_HEAD(&bmsafemap->sm_allocdirecthd, adp, ad_deps);
	}
	LIST_REMOVE(newblk, nb_hash);
	pool_put(&newblk_pool, newblk);

	if (bp == NULL) {
		/*
		 * XXXUBC - Yes, I know how to fix this, but not right now.
		 */
		panic("softdep_setup_allocdirect: Bonk art in the head");
	}
	WORKLIST_INSERT(&bp->b_dep, &adp->ad_list);
	if (lbn >= NDADDR) {
		/* allocating an indirect block */
		if (oldblkno != 0) {
			FREE_LOCK(&lk);
			panic("softdep_setup_allocdirect: non-zero indir");
		}
	} else {
		/*
		 * Allocating a direct block.
		 *
		 * If we are allocating a directory block, then we must
		 * allocate an associated pagedep to track additions and
		 * deletions.
		 */
		if ((DIP(ip, mode) & IFMT) == IFDIR &&
		    pagedep_lookup(ip, lbn, DEPALLOC, &pagedep) == 0)
			WORKLIST_INSERT(&bp->b_dep, &pagedep->pd_list);
	}
	/*
	 * The list of allocdirects must be kept in sorted and ascending
	 * order so that the rollback routines can quickly determine the
	 * first uncommitted block (the size of the file stored on disk
	 * ends at the end of the lowest committed fragment, or if there
	 * are no fragments, at the end of the highest committed block).
	 * Since files generally grow, the typical case is that the new
	 * block is to be added at the end of the list. We speed this
	 * special case by checking against the last allocdirect in the
	 * list before laboriously traversing the list looking for the
	 * insertion point.
	 */
	adphead = &inodedep->id_newinoupdt;
	oldadp = TAILQ_LAST(adphead, allocdirectlst);
	if (oldadp == NULL || oldadp->ad_lbn <= lbn) {
		/* insert at end of list */
		TAILQ_INSERT_TAIL(adphead, adp, ad_next);
		if (oldadp != NULL && oldadp->ad_lbn == lbn)
			allocdirect_merge(adphead, adp, oldadp);
		FREE_LOCK(&lk);
		return;
	}
	TAILQ_FOREACH(oldadp, adphead, ad_next) {
		if (oldadp->ad_lbn >= lbn)
			break;
	}
	if (oldadp == NULL) {
		FREE_LOCK(&lk);
		panic("softdep_setup_allocdirect: lost entry");
	}
	/* insert in middle of list */
	TAILQ_INSERT_BEFORE(oldadp, adp, ad_next);
	if (oldadp->ad_lbn == lbn)
		allocdirect_merge(adphead, adp, oldadp);
	FREE_LOCK(&lk);
}

/*
 * Replace an old allocdirect dependency with a newer one.
 * This routine must be called with splbio interrupts blocked.
 */
/* head of list holding allocdirects */
/* allocdirect being added */
/* existing allocdirect being checked */
STATIC void
allocdirect_merge(struct allocdirectlst *adphead, struct allocdirect *newadp,
    struct allocdirect *oldadp)
{
	struct worklist *wk;
	struct freefrag *freefrag;
	struct newdirblk *newdirblk;

	splassert(IPL_BIO);

#ifdef DEBUG
	if (lk.lkt_held == -1)
		panic("allocdirect_merge: lock not held");
#endif
	if (newadp->ad_oldblkno != oldadp->ad_newblkno ||
	    newadp->ad_oldsize != oldadp->ad_newsize ||
	    newadp->ad_lbn >= NDADDR) {
		FREE_LOCK(&lk);
		panic("allocdirect_merge: old %lld != new %lld || lbn %lld >= "
		    "%d", (long long)newadp->ad_oldblkno,
		    (long long)oldadp->ad_newblkno, (long long)newadp->ad_lbn,
		    NDADDR);
	}
	newadp->ad_oldblkno = oldadp->ad_oldblkno;
	newadp->ad_oldsize = oldadp->ad_oldsize;
	/*
	 * If the old dependency had a fragment to free or had never
	 * previously had a block allocated, then the new dependency
	 * can immediately post its freefrag and adopt the old freefrag.
	 * This action is done by swapping the freefrag dependencies.
	 * The new dependency gains the old one's freefrag, and the
	 * old one gets the new one and then immediately puts it on
	 * the worklist when it is freed by free_allocdirect. It is
	 * not possible to do this swap when the old dependency had a
	 * non-zero size but no previous fragment to free. This condition
	 * arises when the new block is an extension of the old block.
	 * Here, the first part of the fragment allocated to the new
	 * dependency is part of the block currently claimed on disk by
	 * the old dependency, so cannot legitimately be freed until the
	 * conditions for the new dependency are fulfilled.
	 */
	if (oldadp->ad_freefrag != NULL || oldadp->ad_oldblkno == 0) {
		freefrag = newadp->ad_freefrag;
		newadp->ad_freefrag = oldadp->ad_freefrag;
		oldadp->ad_freefrag = freefrag;
	}
	/*
	 * If we are tracking a new directory-block allocation,
	 * move it from the old allocdirect to the new allocdirect.
	 */
	if ((wk = LIST_FIRST(&oldadp->ad_newdirblk)) != NULL) {
		newdirblk = WK_NEWDIRBLK(wk);
		WORKLIST_REMOVE(&newdirblk->db_list);
		if (LIST_FIRST(&oldadp->ad_newdirblk) != NULL)
			panic("allocdirect_merge: extra newdirblk");
		WORKLIST_INSERT(&newadp->ad_newdirblk, &newdirblk->db_list);
	}
	free_allocdirect(adphead, oldadp, 0);
}

/*
 * Allocate a new freefrag structure if needed.
 */
STATIC struct freefrag *
newfreefrag(struct inode *ip, daddr_t blkno, long size)
{
	struct freefrag *freefrag;
	struct fs *fs;

	if (blkno == 0)
		return (NULL);
	fs = ip->i_fs;
	if (fragnum(fs, blkno) + numfrags(fs, size) > fs->fs_frag)
		panic("newfreefrag: frag size");
	freefrag = pool_get(&freefrag_pool, PR_WAITOK);
	freefrag->ff_list.wk_type = D_FREEFRAG;
	freefrag->ff_state = DIP(ip, uid) & ~ONWORKLIST; /* used below */
	freefrag->ff_inum = ip->i_number;
	freefrag->ff_mnt = ITOV(ip)->v_mount;
	freefrag->ff_devvp = ip->i_devvp;
	freefrag->ff_blkno = blkno;
	freefrag->ff_fragsize = size;
	return (freefrag);
}

/*
 * This workitem de-allocates fragments that were replaced during
 * file block allocation.
 */
STATIC void
handle_workitem_freefrag(struct freefrag *freefrag)
{
	struct inode tip;
	struct ufs1_dinode dtip1;

	tip.i_vnode = NULL;
	tip.i_din1 = &dtip1;
	tip.i_fs = VFSTOUFS(freefrag->ff_mnt)->um_fs;
	tip.i_ump = VFSTOUFS(freefrag->ff_mnt);
	tip.i_dev = freefrag->ff_devvp->v_rdev;
	tip.i_number = freefrag->ff_inum;
	tip.i_ffs1_uid = freefrag->ff_state & ~ONWORKLIST; /* set above */
	ffs_blkfree(&tip, freefrag->ff_blkno, freefrag->ff_fragsize);
	pool_put(&freefrag_pool, freefrag);
}

/*
 * Indirect block allocation dependencies.
 *
 * The same dependencies that exist for a direct block also exist when
 * a new block is allocated and pointed to by an entry in a block of
 * indirect pointers. The undo/redo states described above are also
 * used here. Because an indirect block contains many pointers that
 * may have dependencies, a second copy of the entire in-memory indirect
 * block is kept. The buffer cache copy is always completely up-to-date.
 * The second copy, which is used only as a source for disk writes,
 * contains only the safe pointers (i.e., those that have no remaining
 * update dependencies). The second copy is freed when all pointers
 * are safe. The cache is not allowed to replace indirect blocks with
 * pending update dependencies. If a buffer containing an indirect
 * block with dependencies is written, these routines will mark it
 * dirty again. It can only be successfully written once all the
 * dependencies are removed. The ffs_fsync routine in conjunction with
 * softdep_sync_metadata work together to get all the dependencies
 * removed so that a file can be successfully written to disk. Three
 * procedures are used when setting up indirect block pointer
 * dependencies. The division is necessary because of the organization
 * of the "balloc" routine and because of the distinction between file
 * pages and file metadata blocks.
 */

/*
 * Allocate a new allocindir structure.
 */
/* inode for file being extended */
/* offset of pointer in indirect block */
/* disk block number being added */
/* previous block number, 0 if none */
STATIC struct allocindir *
newallocindir(struct inode *ip, int ptrno, daddr_t newblkno,
    daddr_t oldblkno)
{
	struct allocindir *aip;

	aip = pool_get(&allocindir_pool, PR_WAITOK | PR_ZERO);
	aip->ai_list.wk_type = D_ALLOCINDIR;
	aip->ai_state = ATTACHED;
	aip->ai_offset = ptrno;
	aip->ai_newblkno = newblkno;
	aip->ai_oldblkno = oldblkno;
	aip->ai_freefrag = newfreefrag(ip, oldblkno, ip->i_fs->fs_bsize);
	return (aip);
}

/*
 * Called just before setting an indirect block pointer
 * to a newly allocated file page.
 */
/* inode for file being extended */
/* allocated block number within file */
/* buffer with indirect blk referencing page */
/* offset of pointer in indirect block */
/* disk block number being added */
/* previous block number, 0 if none */
/* buffer holding allocated page */
void
softdep_setup_allocindir_page(struct inode *ip, daddr_t lbn, struct buf *bp,
    int ptrno, daddr_t newblkno, daddr_t oldblkno, struct buf *nbp)
{
	struct allocindir *aip;
	struct pagedep *pagedep;

	aip = newallocindir(ip, ptrno, newblkno, oldblkno);
	ACQUIRE_LOCK(&lk);
	/*
	 * If we are allocating a directory page, then we must
	 * allocate an associated pagedep to track additions and
	 * deletions.
	 */
	if ((DIP(ip, mode) & IFMT) == IFDIR &&
	    pagedep_lookup(ip, lbn, DEPALLOC, &pagedep) == 0)
		WORKLIST_INSERT(&nbp->b_dep, &pagedep->pd_list);
	if (nbp == NULL) {
		/*
		 * XXXUBC - Yes, I know how to fix this, but not right now.
		 */
		panic("softdep_setup_allocindir_page: Bonk art in the head");
	}
	WORKLIST_INSERT(&nbp->b_dep, &aip->ai_list);
	FREE_LOCK(&lk);
	setup_allocindir_phase2(bp, ip, aip);
}

/*
 * Called just before setting an indirect block pointer to a
 * newly allocated indirect block.
 */
/* newly allocated indirect block */
/* inode for file being extended */
/* indirect block referencing allocated block */
/* offset of pointer in indirect block */
/* disk block number being added */
void
softdep_setup_allocindir_meta(struct buf *nbp, struct inode *ip,
    struct buf *bp, int ptrno, daddr_t newblkno)
{
	struct allocindir *aip;

	aip = newallocindir(ip, ptrno, newblkno, 0);
	ACQUIRE_LOCK(&lk);
	WORKLIST_INSERT(&nbp->b_dep, &aip->ai_list);
	FREE_LOCK(&lk);
	setup_allocindir_phase2(bp, ip, aip);
}

/*
 * Called to finish the allocation of the "aip" allocated
 * by one of the two routines above.
 */
/* in-memory copy of the indirect block */
/* inode for file being extended */
/* allocindir allocated by the above routines */
STATIC void
setup_allocindir_phase2(struct buf *bp, struct inode *ip,
    struct allocindir *aip)
{
	struct worklist *wk;
	struct indirdep *indirdep, *newindirdep;
	struct bmsafemap *bmsafemap;
	struct allocindir *oldaip;
	struct freefrag *freefrag;
	struct newblk *newblk;

	if (bp->b_lblkno >= 0)
		panic("setup_allocindir_phase2: not indir blk");
	for (indirdep = NULL, newindirdep = NULL; ; ) {
		ACQUIRE_LOCK(&lk);
		LIST_FOREACH(wk, &bp->b_dep, wk_list) {
			if (wk->wk_type != D_INDIRDEP)
				continue;
			indirdep = WK_INDIRDEP(wk);
			break;
		}
		if (indirdep == NULL && newindirdep) {
			indirdep = newindirdep;
			WORKLIST_INSERT(&bp->b_dep, &indirdep->ir_list);
			newindirdep = NULL;
		}
		FREE_LOCK(&lk);
		if (indirdep) {
			if (newblk_lookup(ip->i_fs, aip->ai_newblkno, 0,
			    &newblk) == 0)
				panic("setup_allocindir: lost block");
			ACQUIRE_LOCK(&lk);
			if (newblk->nb_state == DEPCOMPLETE) {
				aip->ai_state |= DEPCOMPLETE;
				aip->ai_buf = NULL;
			} else {
				bmsafemap = newblk->nb_bmsafemap;
				aip->ai_buf = bmsafemap->sm_buf;
				LIST_REMOVE(newblk, nb_deps);
				LIST_INSERT_HEAD(&bmsafemap->sm_allocindirhd,
				    aip, ai_deps);
			}
			LIST_REMOVE(newblk, nb_hash);
			pool_put(&newblk_pool, newblk);
			aip->ai_indirdep = indirdep;
			/*
			 * Check to see if there is an existing dependency
			 * for this block. If there is, merge the old
			 * dependency into the new one.
			 */
			if (aip->ai_oldblkno == 0)
				oldaip = NULL;
			else

				LIST_FOREACH(oldaip, &indirdep->ir_deplisthd, ai_next)
					if (oldaip->ai_offset == aip->ai_offset)
						break;
			freefrag = NULL;
			if (oldaip != NULL) {
				if (oldaip->ai_newblkno != aip->ai_oldblkno) {
					FREE_LOCK(&lk);
					panic("setup_allocindir_phase2: blkno");
				}
				aip->ai_oldblkno = oldaip->ai_oldblkno;
				freefrag = aip->ai_freefrag;
				aip->ai_freefrag = oldaip->ai_freefrag;
				oldaip->ai_freefrag = NULL;
				free_allocindir(oldaip, NULL);
			}
			LIST_INSERT_HEAD(&indirdep->ir_deplisthd, aip, ai_next);
			if (ip->i_ump->um_fstype == UM_UFS1)
				((int32_t *)indirdep->ir_savebp->b_data)
				    [aip->ai_offset] = aip->ai_oldblkno;
			else
				((int64_t *)indirdep->ir_savebp->b_data)
				    [aip->ai_offset] = aip->ai_oldblkno;
			FREE_LOCK(&lk);
			if (freefrag != NULL)
				handle_workitem_freefrag(freefrag);
		}
		if (newindirdep) {
			if (indirdep->ir_savebp != NULL)
				brelse(newindirdep->ir_savebp);
			WORKITEM_FREE(newindirdep, D_INDIRDEP);
		}
		if (indirdep)
			break;
		newindirdep = pool_get(&indirdep_pool, PR_WAITOK);
		newindirdep->ir_list.wk_type = D_INDIRDEP;
		newindirdep->ir_state = ATTACHED;
		if (ip->i_ump->um_fstype == UM_UFS1)
			newindirdep->ir_state |= UFS1FMT;
		LIST_INIT(&newindirdep->ir_deplisthd);
		LIST_INIT(&newindirdep->ir_donehd);
		if (bp->b_blkno == bp->b_lblkno) {
			VOP_BMAP(bp->b_vp, bp->b_lblkno, NULL, &bp->b_blkno,
				NULL);
		}
		newindirdep->ir_savebp =
		    getblk(ip->i_devvp, bp->b_blkno, bp->b_bcount, 0, INFSLP);
#if 0
		BUF_KERNPROC(newindirdep->ir_savebp);
#endif
		memcpy(newindirdep->ir_savebp->b_data, bp->b_data, bp->b_bcount);
	}
}

/*
 * Block de-allocation dependencies.
 *
 * When blocks are de-allocated, the on-disk pointers must be nullified before
 * the blocks are made available for use by other files.  (The true
 * requirement is that old pointers must be nullified before new on-disk
 * pointers are set.  We chose this slightly more stringent requirement to
 * reduce complexity.) Our implementation handles this dependency by updating
 * the inode (or indirect block) appropriately but delaying the actual block
 * de-allocation (i.e., freemap and free space count manipulation) until
 * after the updated versions reach stable storage.  After the disk is
 * updated, the blocks can be safely de-allocated whenever it is convenient.
 * This implementation handles only the common case of reducing a file's
 * length to zero. Other cases are handled by the conventional synchronous
 * write approach.
 *
 * The ffs implementation with which we worked double-checks
 * the state of the block pointers and file size as it reduces
 * a file's length.  Some of this code is replicated here in our
 * soft updates implementation.  The freeblks->fb_chkcnt field is
 * used to transfer a part of this information to the procedure
 * that eventually de-allocates the blocks.
 *
 * This routine should be called from the routine that shortens
 * a file's length, before the inode's size or block pointers
 * are modified. It will save the block pointer information for
 * later release and zero the inode so that the calling routine
 * can release it.
 */
/* The inode whose length is to be reduced */
/* The new length for the file */
void
softdep_setup_freeblocks(struct inode *ip, off_t length)
{
	struct freeblks *freeblks;
	struct inodedep *inodedep;
	struct allocdirect *adp;
	struct vnode *vp;
	struct buf *bp;
	struct fs *fs;
	int i, delay, error;

	fs = ip->i_fs;
	if (length != 0)
		panic("softdep_setup_freeblocks: non-zero length");
	freeblks = pool_get(&freeblks_pool, PR_WAITOK | PR_ZERO);
	freeblks->fb_list.wk_type = D_FREEBLKS;
	freeblks->fb_state = ATTACHED;
	freeblks->fb_uid = DIP(ip, uid);
	freeblks->fb_previousinum = ip->i_number;
	freeblks->fb_devvp = ip->i_devvp;
	freeblks->fb_mnt = ITOV(ip)->v_mount;
	freeblks->fb_oldsize = DIP(ip, size);
	freeblks->fb_newsize = length;
```

```
        freeblks->fb_chkcnt = DIP(ip, blocks);

        for (i = 0; i < NDADDR; i++) {
                freeblks->fb_dblks[i] = DIP(ip, db[i]);
                DIP_ASSIGN(ip, db[i], 0);
        }

        for (i = 0; i < NIADDR; i++) {
                freeblks->fb_iblks[i] = DIP(ip, ib[i]);
                DIP_ASSIGN(ip, ib[i], 0);
        }

        DIP_ASSIGN(ip, blocks, 0);
        DIP_ASSIGN(ip, size, 0);

        /*
         * Push the zero'ed inode to to its disk buffer so that we are free
         * to delete its dependencies below. Once the dependencies are gone
         * the buffer can be safely released.
         */
        if ((error = bread(ip->i_devvp,
            fsbtodb(fs, ino_to_fsba(fs, ip->i_number)),
            (int)fs->fs_bsize, &bp)) != 0)
                softdep_error("softdep_setup_freeblocks", error);

        if (ip->i_ump->um_fstype == UM_UFS1)
                *((struct ufs1_dinode *) bp->b_data +
                    ino_to_fsbo(fs, ip->i_number)) = *ip->i_din1;
        else
                *((struct ufs2_dinode *) bp->b_data +
                    ino_to_fsbo(fs, ip->i_number)) = *ip->i_din2;

        /*
         * Find and eliminate any inode dependencies.
         */
        ACQUIRE_LOCK(&lk);
        (void) inodedep_lookup(fs, ip->i_number, DEPALLOC, &inodedep);
        if ((inodedep->id_state & IOSTARTED) != 0) {
                FREE_LOCK(&lk);
                panic("softdep_setup_freeblocks: inode busy");
        }
        /*
         * Add the freeblks structure to the list of operations that
         * must await the zero'ed inode being written to disk. If we
         * still have a bitmap dependency (delay == 0), then the inode
         * has never been written to disk, so we can process the
         * freeblks below once we have deleted the dependencies.
         */
        delay = (inodedep->id_state & DEPCOMPLETE);
        if (delay)
                WORKLIST_INSERT(&inodedep->id_bufwait, &freeblks->fb_list);
        /*
         * Because the file length has been truncated to zero, any
         * pending block allocation dependency structures associated
         * with this inode are obsolete and can simply be de-allocated.
         * We must first merge the two dependency lists to get rid of
         * any duplicate freefrag structures, then purge the merged list.
         * If we still have a bitmap dependency, then the inode has never
         * been written to disk, so we can free any fragments without delay.
         */
        merge_inode_lists(inodedep);
        while ((adp = TAILQ_FIRST(&inodedep->id_inoupdt)) != NULL)
                free_allocdirect(&inodedep->id_inoupdt, adp, delay);
        FREE_LOCK(&lk);
        bdwrite(bp);
        /*
         * We must wait for any I/O in progress to finish so that
         * all potential buffers on the dirty list will be visible.
         * Once they are all there, walk the list and get rid of
         * any dependencies.
         */
        vp = ITOV(ip);
        ACQUIRE_LOCK(&lk);
        drain_output(vp, 1);
        while ((bp = LIST_FIRST(&vp->v_dirtyblkhd))) {
                if (getdirtybuf(bp, MNT_WAIT) <= 0)
                        break;
                (void) inodedep_lookup(fs, ip->i_number, 0, &inodedep);
                deallocate_dependencies(bp, inodedep);
                bp->b_flags |= B_INVAL | B_NOCACHE;
                FREE_LOCK(&lk);
                brelse(bp);
                ACQUIRE_LOCK(&lk);
        }
        if (inodedep_lookup(fs, ip->i_number, 0, &inodedep) != 0)
                (void) free_inodedep(inodedep);

        if (delay) {
                freeblks->fb_state |= DEPCOMPLETE;
                /*
                 * If the inode with zeroed block pointers is now on disk we
                 * can start freeing blocks. Add freeblks to the worklist
                 * instead of calling handle_workitem_freeblocks() directly as
                 * it is more likely that additional IO is needed to complete
                 * the request than in the !delay case.
                 */
                if ((freeblks->fb_state & ALLCOMPLETE) == ALLCOMPLETE)
                        add_to_worklist(&freeblks->fb_list);
        }

        FREE_LOCK(&lk);
        /*
         * If the inode has never been written to disk (delay == 0),
         * then we can process the freeblks now that we have deleted
         * the dependencies.
         */
        if (!delay)
                handle_workitem_freeblocks(freeblks);
}

/*
 * Reclaim any dependency structures from a buffer that is about to
 * be reallocated to a new vnode. The buffer must be locked, thus,
 * no I/O completion operations can occur while we are manipulating
 * its associated dependencies. The mutex is held so that other I/O's
 * associated with related dependencies do not occur.
 */
STATIC void
deallocate_dependencies(struct buf *bp, struct inodedep *inodedep)
{
        struct worklist *wk;
        struct indirdep *indirdep;
        struct allocindir *aip;
        struct pagedep *pagedep;
        struct dirrem *dirrem;
        struct diradd *dap;
        int i;

        while ((wk = LIST_FIRST(&bp->b_dep)) != NULL) {
                switch (wk->wk_type) {

                case D_INDIRDEP:
                        indirdep = WK_INDIRDEP(wk);
                        /*
                         * None of the indirect pointers will ever be visible,
                         * so they can simply be tossed. GOINGAWAY ensures
                         * that allocated pointers will be saved in the buffer
                         * cache until they are freed. Note that they will
                         * only be able to be found by their physical address
                         * since the inode mapping the logical address will
                         * be gone. The save buffer used for the safe copy
                         * was allocated in setup_allocindir_phase2 using
                         * the physical address so it could be used for this
                         * purpose. Hence we swap the safe copy with the real
                         * copy, allowing the safe copy to be freed and holding
                         * on to the real copy for later use in indir_trunc.
                         */
                        if (indirdep->ir_state & GOINGAWAY) {
                                FREE_LOCK(&lk);
                                panic("deallocate_dependencies: already gone");
                        }
                        indirdep->ir_state |= GOINGAWAY;
                        while ((aip = LIST_FIRST(&indirdep->ir_deplisthd)))
                                free_allocindir(aip, inodedep);
                        if (bp->b_lblkno >= 0 ||
                            bp->b_blkno != indirdep->ir_savebp->b_lblkno) {
                                FREE_LOCK(&lk);
                                panic("deallocate_dependencies: not indir");
                        }
                        memcpy(indirdep->ir_savebp->b_data, bp->b_data,
                            bp->b_bcount);
                        WORKLIST_REMOVE(wk);
                        WORKLIST_INSERT(&indirdep->ir_savebp->b_dep, wk);
                        continue;

                case D_PAGEDEP:
                        pagedep = WK_PAGEDEP(wk);
                        /*
                         * None of the directory additions will ever be
                         * visible, so they can simply be tossed.
                         */
                        for (i = 0; i < DAHASHSZ; i++)
                                while ((dap =
                                    LIST_FIRST(&pagedep->pd_diraddhd[i])))
                                        free_diradd(dap);
                        while ((dap = LIST_FIRST(&pagedep->pd_pendinghd)))
                                free_diradd(dap);
                        /*
                         * Copy any directory remove dependencies to the list
                         * to be processed after the zero'ed inode is written.
                         * If the inode has already been written, then they
                         * can be dumped directly onto the work list.
                         */
                        while ((dirrem = LIST_FIRST(&pagedep->pd_dirremhd))) {
                                LIST_REMOVE(dirrem, dm_next);
                                dirrem->dm_dirinum = pagedep->pd_ino;
                                if (inodedep == NULL ||
                                    (inodedep->id_state & ALLCOMPLETE) ==
                                     ALLCOMPLETE)
                                        add_to_worklist(&dirrem->dm_list);
                                else
                                        WORKLIST_INSERT(&inodedep->id_bufwait,
                                            &dirrem->dm_list);
                        }
                        if ((pagedep->pd_state & NEWBLOCK) != 0) {
                                LIST_FOREACH(wk, &inodedep->id_bufwait, wk_list)
                                        if (wk->wk_type == D_NEWDIRBLK &&
                                            WK_NEWDIRBLK(wk)->db_pagedep ==
                                            pagedep)
                                                break;
                                if (wk != NULL) {
                                        WORKLIST_REMOVE(wk);
                                        free_newdirblk(WK_NEWDIRBLK(wk));
                                } else {
                                        FREE_LOCK(&lk);
                                        panic("deallocate_dependencies: "
                                            "lost pagedep");
                                        }
                        }
                        WORKLIST_REMOVE(&pagedep->pd_list);
                        LIST_REMOVE(pagedep, pd_hash);
                        WORKITEM_FREE(pagedep, D_PAGEDEP);
                        continue;

                case D_ALLOCINDIR:
                        free_allocindir(WK_ALLOCINDIR(wk), inodedep);
                        continue;

                case D_ALLOCDIRECT:
                case D_INODEDEP:
                        FREE_LOCK(&lk);
                        panic("deallocate_dependencies: Unexpected type %s",
                            TYPENAME(wk->wk_type));
                        /* NOTREACHED */

                default:
                        FREE_LOCK(&lk);
                        panic("deallocate_dependencies: Unknown type %s",
                            TYPENAME(wk->wk_type));
                        /* NOTREACHED */
                }
        }
}

/*
 * Free an allocdirect. Generate a new freefrag work request if appropriate.
 * This routine must be called with splbio interrupts blocked.
 */
STATIC void
free_allocdirect(struct allocdirectlst *adphead, struct allocdirect *adp,
    int delay)
{
        struct newdirblk *newdirblk;
        struct worklist *wk;

        splassert(IPL_BIO);

#ifdef DEBUG
        if (lk.lkt_held == -1)
                panic("free_allocdirect: lock not held");
#endif
        if ((adp->ad_state & DEPCOMPLETE) == 0)
                LIST_REMOVE(adp, ad_deps);
        TAILQ_REMOVE(adphead, adp, ad_next);
        if ((adp->ad_state & COMPLETE) == 0)
                WORKLIST_REMOVE(&adp->ad_list);
        if (adp->ad_freefrag != NULL) {
                if (delay)
                        WORKLIST_INSERT(&adp->ad_inodedep->id_bufwait,
                            &adp->ad_freefrag->ff_list);
                else
                        add_to_worklist(&adp->ad_freefrag->ff_list);
        }
        if ((wk = LIST_FIRST(&adp->ad_newdirblk)) != NULL) {
                newdirblk = WK_NEWDIRBLK(wk);
                WORKLIST_REMOVE(&newdirblk->db_list);
                if (LIST_FIRST(&adp->ad_newdirblk) != NULL)
                        panic("free_allocdirect: extra newdirblk");
                if (delay)
                        WORKLIST_INSERT(&adp->ad_inodedep->id_bufwait,
                            &newdirblk->db_list);
                else
                        free_newdirblk(newdirblk);
        }
        WORKITEM_FREE(adp, D_ALLOCDIRECT);
}

/*
 * Free a newdirblk. Clear the NEWBLOCK flag on its associated pagedep.
 * This routine must be called with splbio interrupts blocked.
 */
void
free_newdirblk(struct newdirblk *newdirblk)
{
        struct pagedep *pagedep;
        struct diradd *dap;
        int i;

        splassert(IPL_BIO);

#ifdef DEBUG
        if (lk.lkt_held == -1)
                panic("free_newdirblk: lock not held");
#endif
        /*
         * If the pagedep is still linked onto the directory buffer
         * dependency chain, then some of the entries on the
         * pd_pendinghd list may not be committed to disk yet. In
         * this case, we will simply clear the NEWBLOCK flag and
         * let the pd_pendinghd list be processed when the pagedep
         * is next written. If the pagedep is no longer on the buffer
         * dependency chain, then all the entries on the pd_pending
         * list are committed to disk and we can free them here.
         */
```

```c
        pagedep = newdirblk->db_pagedep;
        pagedep->pd_state &= ~NEWBLOCK;
        if ((pagedep->pd_state & ONWORKLIST) == 0)
                while ((dap = LIST_FIRST(&pagedep->pd_pendinghd)) != NULL)
                        free_diradd(dap);
        /*
         * If no dependencies remain, the pagedep will be freed.
         */
        for (i = 0; i < DAHASHSZ; i++)
                if (LIST_FIRST(&pagedep->pd_diraddhd[i]) != NULL)
                        break;
        if (i == DAHASHSZ && (pagedep->pd_state & ONWORKLIST) == 0) {
                LIST_REMOVE(pagedep, pd_hash);
                WORKITEM_FREE(pagedep, D_PAGEDEP);
        }
        WORKITEM_FREE(newdirblk, D_NEWDIRBLK);
}

/*
 * Prepare an inode to be freed. The actual free operation is not
 * done until the zero'ed inode has been written to disk.
 */
void
softdep_freefile(struct vnode *pvp, ufsino_t ino, mode_t mode)
{
        struct inode *ip = VTOI(pvp);
        struct inodedep *inodedep;
        struct freefile *freefile;

        /*
         * This sets up the inode de-allocation dependency.
         */
        freefile = pool_get(&freefile_pool, PR_WAITOK);
        freefile->fx_list.wk_type = D_FREEFILE;
        freefile->fx_list.wk_state = 0;
        freefile->fx_mode = mode;
        freefile->fx_oldinum = ino;
        freefile->fx_devvp = ip->i_devvp;
        freefile->fx_mnt = ITOV(ip)->v_mount;

        /*
         * If the inodedep does not exist, then the zero'ed inode has
         * been written to disk. If the allocated inode has never been
         * written to disk, then the on-disk inode is zero'ed. In either
         * case we can free the file immediately.
         */
        ACQUIRE_LOCK(&lk);
        if (inodedep_lookup(ip->i_fs, ino, 0, &inodedep) == 0 ||
            check_inode_unwritten(inodedep)) {
                FREE_LOCK(&lk);
                handle_workitem_freefile(freefile);
                return;
        }
        WORKLIST_INSERT(&inodedep->id_inowait, &freefile->fx_list);
        FREE_LOCK(&lk);
}

/*
 * Check to see if an inode has never been written to disk. If
 * so free the inodedep and return success, otherwise return failure.
 * This routine must be called with splbio interrupts blocked.
 *
 * If we still have a bitmap dependency, then the inode has never
 * been written to disk. Drop the dependency as it is no longer
 * necessary since the inode is being deallocated. We set the
 * ALLCOMPLETE flags since the bitmap now properly shows that the
 * inode is not allocated. Even if the inode is actively being
 * written, it has been rolled back to its zero'ed state, so we
 * are ensured that a zero inode is what is on the disk. For short
 * lived files, this change will usually result in removing all the
 * dependencies from the inode so that it can be freed immediately.
 */
STATIC int
check_inode_unwritten(struct inodedep *inodedep)
{
        splassert(IPL_BIO);

        if ((inodedep->id_state & DEPCOMPLETE) != 0 ||
            LIST_FIRST(&inodedep->id_pendinghd) != NULL ||
            LIST_FIRST(&inodedep->id_bufwait) != NULL ||
            LIST_FIRST(&inodedep->id_inowait) != NULL ||
            TAILQ_FIRST(&inodedep->id_inoupdt) != NULL ||
            TAILQ_FIRST(&inodedep->id_newinoupdt) != NULL ||
            inodedep->id_nlinkdelta != 0)
                return (0);
        inodedep->id_state |= ALLCOMPLETE;
        LIST_REMOVE(inodedep, id_deps);
        inodedep->id_buf = NULL;
        if (inodedep->id_state & ONWORKLIST)
                WORKLIST_REMOVE(&inodedep->id_list);
        if (inodedep->id_savedino1 != NULL) {
                free(inodedep->id_savedino1, M_INODEDEP, inodedep->id_unsize);
                inodedep->id_savedino1 = NULL;
        }
        if (free_inodedep(inodedep) == 0) {
                FREE_LOCK(&lk);
                panic("check_inode_unwritten: busy inode");
        }
        return (1);
}

/*
 * Try to free an inodedep structure. Return 1 if it could be freed.
 */
STATIC int
free_inodedep(struct inodedep *inodedep)
{

        if ((inodedep->id_state & ONWORKLIST) != 0 ||
            (inodedep->id_state & ALLCOMPLETE) != ALLCOMPLETE ||
            LIST_FIRST(&inodedep->id_pendinghd) != NULL ||
            LIST_FIRST(&inodedep->id_bufwait) != NULL ||
            LIST_FIRST(&inodedep->id_inowait) != NULL ||
            TAILQ_FIRST(&inodedep->id_inoupdt) != NULL ||
            TAILQ_FIRST(&inodedep->id_newinoupdt) != NULL ||
            inodedep->id_nlinkdelta != 0 || inodedep->id_savedino1 != NULL)
                return (0);
        LIST_REMOVE(inodedep, id_hash);
        WORKITEM_FREE(inodedep, D_INODEDEP);
        num_inodedep -= 1;
        return (1);
}

/*
 * This workitem routine performs the block de-allocation.
 * The workitem is added to the pending list after the updated
 * inode block has been written to disk.  As mentioned above,
 * checks regarding the number of blocks de-allocated (compared
 * to the number of blocks allocated for the file) are also
 * performed in this function.
 */
STATIC void
handle_workitem_freeblocks(struct freeblks *freeblks)
{
        struct inode tip;
        daddr_t bn;
        union {
                struct ufs1_dinode di1;
                struct ufs2_dinode di2;
        } di;
        struct fs *fs;
        int i, level, bsize;
        long nblocks, blocksreleased = 0;
        int error, allerror = 0;
        daddr_t baselbns[NIADDR], tmpval;

        if (VFSTOUFS(freeblks->fb_mnt)->um_fstype == UM_UFS1)
                tip.i_din1 = &di.di1;
        else
                tip.i_din2 = &di.di2;

        tip.i_fs = fs = VFSTOUFS(freeblks->fb_mnt)->um_fs;
        tip.i_number = freeblks->fb_previousinum;
        tip.i_ump = VFSTOUFS(freeblks->fb_mnt);
        tip.i_dev = freeblks->fb_devvp->v_rdev;
        DIP_ASSIGN(&tip, size, freeblks->fb_oldsize);
        DIP_ASSIGN(&tip, uid, freeblks->fb_uid);
        tip.i_vnode = NULL;
        tmpval = 1;
        baselbns[0] = NDADDR;
        for (i = 1; i < NIADDR; i++) {
                tmpval *= NINDIR(fs);
                baselbns[i] = baselbns[i - 1] + tmpval;
        }
        nblocks = btodb(fs->fs_bsize);
        blocksreleased = 0;
        /*
         * Indirect blocks first.
         */
        for (level = (NIADDR - 1); level >= 0; level--) {
                if ((bn = freeblks->fb_iblks[level]) == 0)
                        continue;
                if ((error = indir_trunc(&tip, fsbtodb(fs, bn), level,
                    baselbns[level], &blocksreleased)) != 0)
                        allerror = error;
                ffs_blkfree(&tip, bn, fs->fs_bsize);
                blocksreleased += nblocks;
        }
        /*
         * All direct blocks or frags.
         */
        for (i = (NDADDR - 1); i >= 0; i--) {
                if ((bn = freeblks->fb_dblks[i]) == 0)
                        continue;
                bsize = blksize(fs, &tip, i);
                ffs_blkfree(&tip, bn, bsize);
                blocksreleased += btodb(bsize);
        }

#ifdef DIAGNOSTIC
        if (freeblks->fb_chkcnt != blocksreleased)
                printf("handle_workitem_freeblocks: block count\n");
        if (allerror)
                softdep_error("handle_workitem_freeblks", allerror);
#endif /* DIAGNOSTIC */
        WORKITEM_FREE(freeblks, D_FREEBLKS);
}

/*
 * Release blocks associated with the inode ip and stored in the indirect
 * block dbn. If level is greater than SINGLE, the block is an indirect block
 * and recursive calls to indirtrunc must be used to cleanse other indirect
 * blocks.
 */
STATIC int
indir_trunc(struct inode *ip, daddr_t dbn, int level, daddr_t lbn,
    long *countp)
{
        struct buf *bp;
        int32_t *bap1 = NULL;
        int64_t nb, *bap2 = NULL;
        struct fs *fs;
        struct worklist *wk;
        struct indirdep *indirdep;
        int i, lbnadd, nblocks, ufs1fmt;
        int error, allerror = 0;

        fs = ip->i_fs;
        lbnadd = 1;
        for (i = level; i > 0; i--)
                lbnadd *= NINDIR(fs);
        /*
         * Get buffer of block pointers to be freed. This routine is not
         * called until the zero'ed inode has been written, so it is safe
         * to free blocks as they are encountered. Because the inode has
         * been zero'ed, calls to bmap on these blocks will fail. So, we
         * have to use the on-disk address and the block device for the
         * filesystem to look them up. If the file was deleted before its
         * indirect blocks were all written to disk, the routine that set
         * us up (deallocate_dependencies) will have arranged to leave
         * a complete copy of the indirect block in memory for our use.
         * Otherwise we have to read the blocks in from the disk.
         */
        ACQUIRE_LOCK(&lk);
        if ((bp = incore(ip->i_devvp, dbn)) != NULL &&
            (wk = LIST_FIRST(&bp->b_dep)) != NULL) {
                if (wk->wk_type != D_INDIRDEP ||
                    (indirdep = WK_INDIRDEP(wk))->ir_savebp != bp ||
                    (indirdep->ir_state & GOINGAWAY) == 0) {
                        FREE_LOCK(&lk);
                        panic("indir_trunc: lost indirdep");
                }
                WORKLIST_REMOVE(wk);
                WORKITEM_FREE(indirdep, D_INDIRDEP);
                if (LIST_FIRST(&bp->b_dep) != NULL) {
                        FREE_LOCK(&lk);
                        panic("indir_trunc: dangling dep");
                }
                FREE_LOCK(&lk);
        } else {
                FREE_LOCK(&lk);
                error = bread(ip->i_devvp, dbn, (int)fs->fs_bsize, &bp);
                if (error)
                        return (error);
        }
        /*
         * Recursively free indirect blocks.
         */
        if (ip->i_ump->um_fstype == UM_UFS1) {
                ufs1fmt = 1;
                bap1 = (int32_t *)bp->b_data;
        } else {
                ufs1fmt = 0;
                bap2 = (int64_t *)bp->b_data;
        }
        nblocks = btodb(fs->fs_bsize);
        for (i = NINDIR(fs) - 1; i >= 0; i--) {
                if (ufs1fmt)
                        nb = bap1[i];
                else
                        nb = bap2[i];
                if (nb == 0)
                        continue;
                if (level != 0) {
                        if ((error = indir_trunc(ip, fsbtodb(fs, nb),
                            level - 1, lbn + (i * lbnadd), countp)) != 0)
                                allerror = error;
                }
                ffs_blkfree(ip, nb, fs->fs_bsize);
                *countp += nblocks;
        }
        bp->b_flags |= B_INVAL | B_NOCACHE;
        brelse(bp);
        return (allerror);
}

/*
 * Free an allocindir.
 * This routine must be called with splbio interrupts blocked.
 */
STATIC void
free_allocindir(struct allocindir *aip, struct inodedep *inodedep)
{
        struct freefrag *freefrag;

        splassert(IPL_BIO);

#ifdef DEBUG
        if (lk.lkt_held == -1)
                panic("free_allocindir: lock not held");
#endif
        if ((aip->ai_state & DEPCOMPLETE) == 0)
                LIST_REMOVE(aip, ai_deps);
        if (aip->ai_state & ONWORKLIST)
```

```
                WORKLIST_REMOVE(&aip->ai_list);
        LIST_REMOVE(aip, ai_next);
        if ((freefrag = aip->ai_freefrag) != NULL) {
                if (inodedep == NULL)
                        add_to_worklist(&freefrag->ff_list);
                else
                        WORKLIST_INSERT(&inodedep->id_bufwait,
                            &freefrag->ff_list);
        }
        WORKITEM_FREE(aip, D_ALLOCINDIR);
}

/*
 * Directory entry addition dependencies.
 *
 * When adding a new directory entry, the inode (with its incremented link
 * count) must be written to disk before the directory entry's pointer to it.
 * Also, if the inode is newly allocated, the corresponding freemap must be
 * updated (on disk) before the directory entry's pointer. These requirements
 * are met via undo/redo on the directory entry's pointer, which consists
 * simply of the inode number.
 *
 * As directory entries are added and deleted, the free space within a
 * directory block can become fragmented.  The ufs file system will compact
 * a fragmented directory block to make space for a new entry. When this
 * occurs, the offsets of previously added entries change. Any "diradd"
 * dependency structures corresponding to these entries must be updated with
 * the new offsets.
 */

/*
 * This routine is called after the in-memory inode's link
 * count has been incremented, but before the directory entry's
 * pointer to the inode has been set.
 */
/* buffer containing directory block */
/* inode for directory */
/* offset of new entry in directory */
/* inode referenced by new directory entry */
/* non-NULL => contents of new mkdir */
/* entry is in a newly allocated block */
int
softdep_setup_directory_add(struct buf *bp, struct inode *dp, off_t diroffset,
    long newinum, struct buf *newdirbp, int isnewblk)
{
        int offset;             /* offset of new entry within directory block */
        daddr_t lbn;            /* block in directory containing new entry */
        struct fs *fs;
        struct diradd *dap;
        struct allocdirect *adp;
        struct pagedep *pagedep;
        struct inodedep *inodedep;
        struct newdirblk *newdirblk = NULL;
        struct mkdir *mkdir1, *mkdir2;


        fs = dp->i_fs;
        lbn = lblkno(fs, diroffset);
        offset = blkoff(fs, diroffset);
        dap = pool_get(&diradd_pool, PR_WAITOK | PR_ZERO);
        dap->da_list.wk_type = D_DIRADD;
        dap->da_offset = offset;
        dap->da_newinum = newinum;
        dap->da_state = ATTACHED;
        if (isnewblk && lbn < NDADDR && fragoff(fs, diroffset) == 0) {
                newdirblk = pool_get(&newdirblk_pool, PR_WAITOK);
                newdirblk->db_list.wk_type = D_NEWDIRBLK;
                newdirblk->db_state = 0;
        }
        if (newdirbp == NULL) {
                dap->da_state |= DEPCOMPLETE;
                ACQUIRE_LOCK(&lk);
        } else {
                dap->da_state |= MKDIR_BODY | MKDIR_PARENT;
                mkdir1 = pool_get(&mkdir_pool, PR_WAITOK);
                mkdir1->md_list.wk_type = D_MKDIR;
                mkdir1->md_state = MKDIR_BODY;
                mkdir1->md_diradd = dap;
                mkdir2 = pool_get(&mkdir_pool, PR_WAITOK);
                mkdir2->md_list.wk_type = D_MKDIR;
                mkdir2->md_state = MKDIR_PARENT;
                mkdir2->md_diradd = dap;
                /*
                 * Dependency on "." and ".." being written to disk.
                 */
                mkdir1->md_buf = newdirbp;
                ACQUIRE_LOCK(&lk);
                LIST_INSERT_HEAD(&mkdirlisthd, mkdir1, md_mkdirs);
                WORKLIST_INSERT(&newdirbp->b_dep, &mkdir1->md_list);
                FREE_LOCK(&lk);
                bdwrite(newdirbp);
                /*
                 * Dependency on link count increase for parent directory
                 */
                ACQUIRE_LOCK(&lk);
                if (inodedep_lookup(fs, dp->i_number, 0, &inodedep) == 0
                    || (inodedep->id_state & ALLCOMPLETE) == ALLCOMPLETE) {
                        dap->da_state &= ~MKDIR_PARENT;
                        WORKITEM_FREE(mkdir2, D_MKDIR);
                } else {
                        LIST_INSERT_HEAD(&mkdirlisthd, mkdir2, md_mkdirs);
                        WORKLIST_INSERT(&inodedep->id_bufwait,&mkdir2->md_list);
                }
        }
        /*
         * Link into parent directory pagedep to await its being written.
         */
        if (pagedep_lookup(dp, lbn, DEPALLOC, &pagedep) == 0)
                WORKLIST_INSERT(&bp->b_dep, &pagedep->pd_list);
        dap->da_pagedep = pagedep;
        LIST_INSERT_HEAD(&pagedep->pd_diraddhd[DIRADDHASH(offset)], dap,
            da_pdlist);
        /*
         * Link into its inodedep. Put it on the id_bufwait list if the inode
         * is not yet written. If it is written, do the post-inode write
         * processing to put it on the id_pendinghd list.
         */
        (void) inodedep_lookup(fs, newinum, DEPALLOC, &inodedep);
        if ((inodedep->id_state & ALLCOMPLETE) == ALLCOMPLETE)
                diradd_inode_written(dap, inodedep);
        else
                WORKLIST_INSERT(&inodedep->id_bufwait, &dap->da_list);
        if (isnewblk) {
                /*
                 * Directories growing into indirect blocks are rare
                 * enough and the frequency of new block allocation
                 * in those cases even more rare, that we choose not
                 * to bother tracking them. Rather we simply force the
                 * new directory entry to disk.
                 */
                if (lbn >= NDADDR) {
                        FREE_LOCK(&lk);
                        /*
                         * We only have a new allocation when at the
                         * beginning of a new block, not when we are
                         * expanding into an existing block.
                         */
                        if (blkoff(fs, diroffset) == 0)
                                return (1);
                        return (0);
                }
                /*
                 * We only have a new allocation when at the beginning
                 * of a new fragment, not when we are expanding into an
                 * existing fragment. Also, there is nothing to do if we
                 * are already tracking this block.
                 */
                if (fragoff(fs, diroffset) != 0) {
                        FREE_LOCK(&lk);
                        return (0);
                }

                if ((pagedep->pd_state & NEWBLOCK) != 0) {
                        WORKITEM_FREE(newdirblk, D_NEWDIRBLK);
                        FREE_LOCK(&lk);
                        return (0);
                }
                /*
                 * Find our associated allocdirect and have it track us.
                 */
                if (inodedep_lookup(fs, dp->i_number, 0, &inodedep) == 0)
                        panic("softdep_setup_directory_add: lost inodedep");
                adp = TAILQ_LAST(&inodedep->id_newinoupdt, allocdirectlst);
                if (adp == NULL || adp->ad_lbn != lbn) {
                        FREE_LOCK(&lk);
                        panic("softdep_setup_directory_add: lost entry");
                }
                pagedep->pd_state |= NEWBLOCK;
                newdirblk->db_pagedep = pagedep;
                WORKLIST_INSERT(&adp->ad_newdirblk, &newdirblk->db_list);
        }
        FREE_LOCK(&lk);
        return (0);
}

/*
 * This procedure is called to change the offset of a directory
 * entry when compacting a directory block which must be owned
 * exclusively by the caller. Note that the actual entry movement
 * must be done in this procedure to ensure that no I/O completions
 * occur while the move is in progress.
 */
/* inode for directory */
/* address of dp->i_offset */
/* address of old directory location */
/* address of new directory location */
/* size of directory entry */
void
softdep_change_directoryentry_offset(struct inode *dp, caddr_t base,
    caddr_t oldloc, caddr_t newloc, int entrysize)
{
        int offset, oldoffset, newoffset;
        struct pagedep *pagedep;
        struct diradd *dap;
        daddr_t lbn;

        ACQUIRE_LOCK(&lk);
        lbn = lblkno(dp->i_fs, dp->i_offset);
        offset = blkoff(dp->i_fs, dp->i_offset);
        if (pagedep_lookup(dp, lbn, 0, &pagedep) == 0)
                goto done;
        oldoffset = offset + (oldloc - base);
        newoffset = offset + (newloc - base);

        LIST_FOREACH(dap, &pagedep->pd_diraddhd[DIRADDHASH(oldoffset)], da_pdlist) {
                if (dap->da_offset != oldoffset)
                        continue;
                dap->da_offset = newoffset;
                if (DIRADDHASH(newoffset) == DIRADDHASH(oldoffset))
                        break;
                LIST_REMOVE(dap, da_pdlist);
                LIST_INSERT_HEAD(&pagedep->pd_diraddhd[DIRADDHASH(newoffset)],
                    dap, da_pdlist);
                break;
        }
        if (dap == NULL) {

                LIST_FOREACH(dap, &pagedep->pd_pendinghd, da_pdlist) {
                        if (dap->da_offset == oldoffset) {
                                dap->da_offset = newoffset;
                                break;
                        }
                }
        }
done:
        memmove(newloc, oldloc, entrysize);
        FREE_LOCK(&lk);
}

/*
 * Free a diradd dependency structure. This routine must be called
 * with splbio interrupts blocked.
 */
STATIC void
free_diradd(struct diradd *dap)
{
        struct dirrem *dirrem;
        struct pagedep *pagedep;
        struct inodedep *inodedep;
        struct mkdir *mkdir, *nextmd;

        splassert(IPL_BIO);

#ifdef DEBUG
        if (lk.lkt_held == -1)
                panic("free_diradd: lock not held");
#endif
        WORKLIST_REMOVE(&dap->da_list);
        LIST_REMOVE(dap, da_pdlist);
        if ((dap->da_state & DIRCHG) == 0) {
                pagedep = dap->da_pagedep;
        } else {
                dirrem = dap->da_previous;
                pagedep = dirrem->dm_pagedep;
                dirrem->dm_dirinum = pagedep->pd_ino;
                add_to_worklist(&dirrem->dm_list);
        }
        if (inodedep_lookup(VFSTOUFS(pagedep->pd_mnt)->um_fs, dap->da_newinum,
            0, &inodedep) != 0)
                (void) free_inodedep(inodedep);
        if ((dap->da_state & (MKDIR_PARENT | MKDIR_BODY)) != 0) {
                for (mkdir = LIST_FIRST(&mkdirlisthd); mkdir; mkdir = nextmd) {
                        nextmd = LIST_NEXT(mkdir, md_mkdirs);
                        if (mkdir->md_diradd != dap)
                                continue;
                        dap->da_state &= ~mkdir->md_state;
                        WORKLIST_REMOVE(&mkdir->md_list);
                        LIST_REMOVE(mkdir, md_mkdirs);
                        WORKITEM_FREE(mkdir, D_MKDIR);
                }
                if ((dap->da_state & (MKDIR_PARENT | MKDIR_BODY)) != 0) {
                        FREE_LOCK(&lk);
                        panic("free_diradd: unfound ref");
                }
        }
        WORKITEM_FREE(dap, D_DIRADD);
}

/*
 * Directory entry removal dependencies.
 *
 * When removing a directory entry, the entry's inode pointer must be
 * zero'ed on disk before the corresponding inode's link count is decremented
 * (possibly freeing the inode for re-use). This dependency is handled by
 * updating the directory entry but delaying the inode count reduction until
 * after the directory block has been written to disk. After this point, the
 * inode count can be decremented whenever it is convenient.
 */

/*
 * This routine should be called immediately after removing
 * a directory entry.  The inode's link count should not be
 * decremented by the calling procedure -- the soft updates
 * code will do this task when it is safe.
 */
/* buffer containing directory block */
/* inode for the directory being modified */
/* inode for directory entry being removed */
/* indicates if doing RMDIR */
void
softdep_setup_remove(struct buf *bp, struct inode *dp, struct inode *ip,
    int isrmdir)
{
        struct dirrem *dirrem, *prevdirrem;
```

```c
	/*
	 * Allocate a new dirrem if appropriate and ACQUIRE_LOCK.
	 */
	dirrem = newdirrem(bp, dp, ip, isrmdir, &prevdirrem);

	/*
	 * If the COMPLETE flag is clear, then there were no active
	 * entries and we want to roll back to a zeroed entry until
	 * the new inode is committed to disk. If the COMPLETE flag is
	 * set then we have deleted an entry that never made it to
	 * disk. If the entry we deleted resulted from a name change,
	 * then the old name still resides on disk. We cannot delete
	 * its inode (returned to us in prevdirrem) until the zeroed
	 * directory entry gets to disk. The new inode has never been
	 * referenced on the disk, so can be deleted immediately.
	 */
	if ((dirrem->dm_state & COMPLETE) == 0) {
		LIST_INSERT_HEAD(&dirrem->dm_pagedep->pd_dirremhd, dirrem,
		    dm_next);
		FREE_LOCK(&lk);
	} else {
		if (prevdirrem != NULL)
			LIST_INSERT_HEAD(&dirrem->dm_pagedep->pd_dirremhd,
			    prevdirrem, dm_next);
		dirrem->dm_dirinum = dirrem->dm_pagedep->pd_ino;
		FREE_LOCK(&lk);
		handle_workitem_remove(dirrem);
	}
}

STATIC long num_dirrem;		/* number of dirrem allocated */
/*
 * Allocate a new dirrem if appropriate and return it along with
 * its associated pagedep. Called without a lock, returns with lock.
 */
/* buffer containing directory block */
/* inode for the directory being modified */
/* inode for directory entry being removed */
/* indicates if doing RMDIR */
/* previously referenced inode, if any */
STATIC struct dirrem *
newdirrem(struct buf *bp, struct inode *dp, struct inode *ip, int isrmdir,
    struct dirrem **prevdirremp)
{
	int offset;
	daddr_t lbn;
	struct diradd *dap;
	struct dirrem *dirrem;
	struct pagedep *pagedep;

	/*
	 * Whiteouts have no deletion dependencies.
	 */
	if (ip == NULL)
		panic("newdirrem: whiteout");
	/*
	 * If we are over our limit, try to improve the situation.
	 * Limiting the number of dirrem structures will also limit
	 * the number of freefile and freeblks structures.
	 */
	if (num_dirrem > max_softdeps / 2)
		(void) request_cleanup(FLUSH_REMOVE, 0);
	num_dirrem += 1;
	dirrem = pool_get(&dirrem_pool, PR_WAITOK | PR_ZERO);
	dirrem->dm_list.wk_type = D_DIRREM;
	dirrem->dm_state = isrmdir ? RMDIR : 0;
	dirrem->dm_mnt = ITOV(ip)->v_mount;
	dirrem->dm_oldinum = ip->i_number;
	*prevdirremp = NULL;

	ACQUIRE_LOCK(&lk);
	lbn = lblkno(dp->i_fs, dp->i_offset);
	offset = blkoff(dp->i_fs, dp->i_offset);
	if (pagedep_lookup(dp, lbn, DEPALLOC, &pagedep) == 0)
		WORKLIST_INSERT(&bp->b_dep, &pagedep->pd_list);
	dirrem->dm_pagedep = pagedep;
	/*
	 * Check for a diradd dependency for the same directory entry.
	 * If present, then both dependencies become obsolete and can
	 * be de-allocated. Check for an entry on both the pd_dirraddhd
	 * list and the pd_pendinghd list.
	 */

	LIST_FOREACH(dap, &pagedep->pd_diraddhd[DIRADDHASH(offset)], da_pdlist)
		if (dap->da_offset == offset)
			break;
	if (dap == NULL) {

		LIST_FOREACH(dap, &pagedep->pd_pendinghd, da_pdlist)
			if (dap->da_offset == offset)
				break;
		if (dap == NULL)
			return (dirrem);
	}
	/*
	 * Must be ATTACHED at this point.
	 */
	if ((dap->da_state & ATTACHED) == 0) {
		FREE_LOCK(&lk);
		panic("newdirrem: not ATTACHED");
	}
	if (dap->da_newinum != ip->i_number) {
		FREE_LOCK(&lk);
		panic("newdirrem: inum %u should be %u",
		    ip->i_number, dap->da_newinum);
	}
	/*
	 * If we are deleting a changed name that never made it to disk,
	 * then return the dirrem describing the previous inode (which
	 * represents the inode currently referenced from this entry on disk).
	 */
	if ((dap->da_state & DIRCHG) != 0) {
		*prevdirremp = dap->da_previous;
		dap->da_state &= ~DIRCHG;
		dap->da_pagedep = pagedep;
	}
	/*
	 * We are deleting an entry that never made it to disk.
	 * Mark it COMPLETE so we can delete its inode immediately.
	 */
	dirrem->dm_state |= COMPLETE;
	free_diradd(dap);
	return (dirrem);
}

/*
 * Directory entry change dependencies.
 *
 * Changing an existing directory entry requires that an add operation
 * be completed first followed by a deletion. The semantics for the addition
 * are identical to the description of adding a new entry above except
 * that the rollback is to the old inode number rather than zero. Once
 * the addition dependency is completed, the removal is done as described
 * in the removal routine above.
 */

/*
 * This routine should be called immediately after changing
 * a directory entry.  The inode's link count should not be
 * decremented by the calling procedure -- the soft updates
 * code will perform this task when it is safe.
 */
/* buffer containing directory block */
/* inode for the directory being modified */
/* inode for directory entry being removed */
/* new inode number for changed entry */
/* indicates if doing RMDIR */
void
softdep_setup_directory_change(struct buf *bp, struct inode *dp,
    struct inode *ip, long newinum, int isrmdir)
{
	int offset;
	struct diradd *dap;
	struct dirrem *dirrem, *prevdirrem;
	struct pagedep *pagedep;
	struct inodedep *inodedep;

	offset = blkoff(dp->i_fs, dp->i_offset);
	dap = pool_get(&diradd_pool, PR_WAITOK | PR_ZERO);
	dap->da_list.wk_type = D_DIRADD;
	dap->da_state = DIRCHG | ATTACHED | DEPCOMPLETE;
	dap->da_offset = offset;
	dap->da_newinum = newinum;

	/*
	 * Allocate a new dirrem and ACQUIRE_LOCK.
	 */
	dirrem = newdirrem(bp, dp, ip, isrmdir, &prevdirrem);
	pagedep = dirrem->dm_pagedep;
	/*
	 * The possible values for isrmdir:
	 *	0 - non-directory file rename
	 *	1 - directory rename within same directory
	 *   inum - directory rename to new directory of given inode number
	 * When renaming to a new directory, we are both deleting and
	 * creating a new directory entry, so the link count on the new
	 * directory should not change. Thus we do not need the followup
	 * dirrem which is usually done in handle_workitem_remove. We set
	 * the DIRCHG flag to tell handle_workitem_remove to skip the
	 * followup dirrem.
	 */
	if (isrmdir > 1)
		dirrem->dm_state |= DIRCHG;

	/*
	 * If the COMPLETE flag is clear, then there were no active
	 * entries and we want to roll back to the previous inode until
	 * the new inode is committed to disk. If the COMPLETE flag is
	 * set, then we have deleted an entry that never made it to disk.
	 * If the entry we deleted resulted from a name change, then the old
	 * inode reference still resides on disk. Any rollback that we do
	 * needs to be to that old inode (returned to us in prevdirrem). If
	 * the entry we deleted resulted from a create, then there is
	 * no entry on the disk, so we want to roll back to zero rather
	 * than the uncommitted inode. In either of the COMPLETE cases we
	 * want to immediately free the unwritten and unreferenced inode.
	 */
	if ((dirrem->dm_state & COMPLETE) == 0) {
		dap->da_previous = dirrem;
	} else {
		if (prevdirrem != NULL) {
			dap->da_previous = prevdirrem;
		} else {
			dap->da_state &= ~DIRCHG;
			dap->da_pagedep = pagedep;
		}
		dirrem->dm_dirinum = pagedep->pd_ino;
		add_to_worklist(&dirrem->dm_list);
	}
	/*
	 * Link into its inodedep. Put it on the id_bufwait list if the inode
	 * is not yet written. If it is written, do the post-inode write
	 * processing to put it on the id_pendinghd list.
	 */
	if (inodedep_lookup(dp->i_fs, newinum, DEPALLOC, &inodedep) == 0 ||
	    (inodedep->id_state & ALLCOMPLETE) == ALLCOMPLETE) {
		dap->da_state |= COMPLETE;
		LIST_INSERT_HEAD(&pagedep->pd_pendinghd, dap, da_pdlist);
		WORKLIST_INSERT(&inodedep->id_pendinghd, &dap->da_list);
	} else {
		LIST_INSERT_HEAD(&pagedep->pd_diraddhd[DIRADDHASH(offset)],
		    dap, da_pdlist);
		WORKLIST_INSERT(&inodedep->id_bufwait, &dap->da_list);
	}
	FREE_LOCK(&lk);
}

/*
 * Called whenever the link count on an inode is changed.
 * It creates an inode dependency so that the new reference(s)
 * to the inode cannot be committed to disk until the updated
 * inode has been written.
 */
/* the inode with the increased link count */
/* do background work or not */
void
softdep_change_linkcnt(struct inode *ip, int nodelay)
{
	struct inodedep *inodedep;
	int flags;

	/*
	 * If requested, do not allow background work to happen.
	 */
	flags = DEPALLOC;
	if (nodelay)
		flags |= NODELAY;

	ACQUIRE_LOCK(&lk);

	(void) inodedep_lookup(ip->i_fs, ip->i_number, flags, &inodedep);
	if (DIP(ip, nlink) < ip->i_effnlink) {
		FREE_LOCK(&lk);
		panic("softdep_change_linkcnt: bad delta");
	}

	inodedep->id_nlinkdelta = DIP(ip, nlink) - ip->i_effnlink;

	FREE_LOCK(&lk);
}

/*
 * This workitem decrements the inode's link count.
 * If the link count reaches zero, the file is removed.
 */
STATIC void
handle_workitem_remove(struct dirrem *dirrem)
{
	struct proc *p = CURPROC;/* XXX */
	struct inodedep *inodedep;
	struct vnode *vp;
	struct inode *ip;
	ufsino_t oldinum;
	int error;

	if ((error = VFS_VGET(dirrem->dm_mnt, dirrem->dm_oldinum, &vp)) != 0) {
		softdep_error("handle_workitem_remove: vget", error);
		return;
	}
	ip = VTOI(vp);
	ACQUIRE_LOCK(&lk);
	if ((inodedep_lookup(ip->i_fs, dirrem->dm_oldinum, 0, &inodedep))
	    == 0) {
		FREE_LOCK(&lk);
		panic("handle_workitem_remove: lost inodedep");
	}
	/*
	 * Normal file deletion.
	 */
	if ((dirrem->dm_state & RMDIR) == 0) {
		DIP_ADD(ip, nlink, -1);
		ip->i_flag |= IN_CHANGE;
		if (DIP(ip, nlink) < ip->i_effnlink) {
			FREE_LOCK(&lk);
			panic("handle_workitem_remove: bad file delta");
		}
		inodedep->id_nlinkdelta = DIP(ip, nlink) - ip->i_effnlink;
		FREE_LOCK(&lk);
		vput(vp);
		num_dirrem -= 1;
		WORKITEM_FREE(dirrem, D_DIRREM);
		return;
	}
	/*
	 * Directory deletion. Decrement reference count for both the
	 * just deleted parent directory entry and the reference for ".".
	 * Next truncate the directory to length zero. When the
	 * truncation completes, arrange to have the reference count on
	 * the parent decremented to account for the loss of "..".
	 */
	DIP_ADD(ip, nlink, -2);
	ip->i_flag |= IN_CHANGE;
	if (DIP(ip, nlink) < ip->i_effnlink)
		panic("handle_workitem_remove: bad dir delta");
	inodedep->id_nlinkdelta = DIP(ip, nlink) - ip->i_effnlink;
	FREE_LOCK(&lk);
	if ((error = UFS_TRUNCATE(ip, (off_t)0, 0, p->p_ucred)) != 0)
		softdep_error("handle_workitem_remove: truncate", error);
	/*
	 * Rename a directory to a new parent. Since, we are both deleting
	 * and creating a new directory entry, the link count on the new
	 * directory should not change. Thus we skip the followup dirrem.
	 */
	if (dirrem->dm_state & DIRCHG) {
		vput(vp);
		num_dirrem -= 1;
		WORKITEM_FREE(dirrem, D_DIRREM);
		return;
	}
	/*
	 * If the inodedep does not exist, then the zero'ed inode has
	 * been written to disk. If the allocated inode has never been
	 * written to disk, then the on-disk inode is zero'ed. In either
	 * case we can remove the file immediately.
	 */
	ACQUIRE_LOCK(&lk);
	dirrem->dm_state = 0;
	oldinum = dirrem->dm_oldinum;
	dirrem->dm_oldinum = dirrem->dm_dirinum;
	if (inodedep_lookup(ip->i_fs, oldinum, 0, &inodedep) == 0 ||
	    check_inode_unwritten(inodedep)) {
		FREE_LOCK(&lk);
		vput(vp);
		handle_workitem_remove(dirrem);
		return;
	}
	WORKLIST_INSERT(&inodedep->id_inowait, &dirrem->dm_list);
	FREE_LOCK(&lk);
	ip->i_flag |= IN_CHANGE;
	UFS_UPDATE(VTOI(vp), 0);
	vput(vp);
}

/*
 * Inode de-allocation dependencies.
 *
 * When an inode's link count is reduced to zero, it can be de-allocated. We
 * found it convenient to postpone de-allocation until after the inode is
 * written to disk with its new link count (zero).  At this point, all of the
 * on-disk inode's block pointers are nullified and, with careful dependency
 * list ordering, all dependencies related to the inode will be satisfied and
 * the corresponding dependency structures de-allocated.  So, if/when the
 * inode is reused, there will be no mixing of old dependencies with new
 * ones.  This artificial dependency is set up by the block de-allocation
 * procedure above (softdep_setup_freeblocks) and completed by the
 * following procedure.
 */
STATIC void
handle_workitem_freefile(struct freefile *freefile)
{
	struct fs *fs;
	struct vnode vp;
	struct inode tip;
#ifdef DEBUG
	struct inodedep *idp;
#endif
	int error;

	fs = VFSTOUFS(freefile->fx_mnt)->um_fs;
#ifdef DEBUG
	ACQUIRE_LOCK(&lk);
	error = inodedep_lookup(fs, freefile->fx_oldinum, 0, &idp);
	FREE_LOCK(&lk);
	if (error)
		panic("handle_workitem_freefile: inodedep survived");
#endif
	tip.i_ump = VFSTOUFS(freefile->fx_mnt);
	tip.i_dev = freefile->fx_devvp->v_rdev;
	tip.i_fs = fs;
	tip.i_vnode = &vp;
	vp.v_data = &tip;

	if ((error = ffs_freefile(&tip, freefile->fx_oldinum,
		 freefile->fx_mode)) != 0) {
		softdep_error("handle_workitem_freefile", error);
	}
	WORKITEM_FREE(freefile, D_FREEFILE);
}

/*
 * Disk writes.
 *
 * The dependency structures constructed above are most actively used when file
 * system blocks are written to disk.  No constraints are placed on when a
 * block can be written, but unsatisfied update dependencies are made safe by
 * modifying (or replacing) the source memory for the duration of the disk
 * write.  When the disk write completes, the memory block is again brought
 * up-to-date.
 *
 * In-core inode structure reclamation.
 *
 * Because there are a finite number of "in-core" inode structures, they are
 * reused regularly.  By transferring all inode-related dependencies to the
 * in-memory inode block and indexing them separately (via "inodedep"s), we
 * can allow "in-core" inode structures to be reused at any time and avoid
 * any increase in contention.
 *
 * Called just before entering the device driver to initiate a new disk I/O.
 * The buffer must be locked, thus, no I/O completion operations can occur
 * while we are manipulating its associated dependencies.
 */
/* structure describing disk write to occur */
void
softdep_disk_io_initiation(struct buf *bp)
{
	struct worklist *wk, *nextwk;
	struct indirdep *indirdep;
	struct inodedep *inodedep;
	struct buf *sbp;

	/*
	 * We only care about write operations. There should never
	 * be dependencies for reads.
	 */
	if (bp->b_flags & B_READ)
		panic("softdep_disk_io_initiation: read");

	ACQUIRE_LOCK(&lk);

	/*
	 * Do any necessary pre-I/O processing.
	 */
	for (wk = LIST_FIRST(&bp->b_dep); wk; wk = nextwk) {
		nextwk = LIST_NEXT(wk, wk_list);
		switch (wk->wk_type) {

		case D_PAGEDEP:
			initiate_write_filepage(WK_PAGEDEP(wk), bp);
			continue;

		case D_INODEDEP:
			inodedep = WK_INODEDEP(wk);
			if (inodedep->id_fs->fs_magic == FS_UFS1_MAGIC)
				initiate_write_inodeblock_ufs1(inodedep, bp);
#ifdef FFS2
			else
				initiate_write_inodeblock_ufs2(inodedep, bp);
#endif
			continue;

		case D_INDIRDEP:
			indirdep = WK_INDIRDEP(wk);
			if (indirdep->ir_state & GOINGAWAY)
				panic("disk_io_initiation: indirdep gone");
			/*
			 * If there are no remaining dependencies, this
			 * will be writing the real pointers, so the
			 * dependency can be freed.
			 */
			if (LIST_FIRST(&indirdep->ir_deplisthd) == NULL) {
				sbp = indirdep->ir_savebp;
				sbp->b_flags |= B_INVAL | B_NOCACHE;
				/* inline expand WORKLIST_REMOVE(wk); */
				wk->wk_state &= ~ONWORKLIST;
				LIST_REMOVE(wk, wk_list);
				WORKITEM_FREE(indirdep, D_INDIRDEP);
				FREE_LOCK(&lk);
				brelse(sbp);
				ACQUIRE_LOCK(&lk);
				continue;
			}
			/*
			 * Replace up-to-date version with safe version.
			 */
			FREE_LOCK(&lk);
			indirdep->ir_saveddata = malloc(bp->b_bcount,
			    M_INDIRDEP, M_WAITOK);
			ACQUIRE_LOCK(&lk);
			indirdep->ir_state &= ~ATTACHED;
			indirdep->ir_state |= UNDONE;
			memcpy(indirdep->ir_saveddata, bp->b_data, bp->b_bcount);
			memcpy(bp->b_data, indirdep->ir_savebp->b_data,
			    bp->b_bcount);
			continue;

		case D_MKDIR:
		case D_BMSAFEMAP:
		case D_ALLOCDIRECT:
		case D_ALLOCINDIR:
			continue;

		default:
			FREE_LOCK(&lk);
			panic("handle_disk_io_initiation: Unexpected type %s",
			    TYPENAME(wk->wk_type));
			/* NOTREACHED */
		}
	}

	FREE_LOCK(&lk);
}

/*
 * Called from within the procedure above to deal with unsatisfied
 * allocation dependencies in a directory. The buffer must be locked,
 * thus, no I/O completion operations can occur while we are
 * manipulating its associated dependencies.
 */
STATIC void
initiate_write_filepage(struct pagedep *pagedep, struct buf *bp)
{
	struct diradd *dap;
	struct direct *ep;
	int i;

	if (pagedep->pd_state & IOSTARTED) {
		/*
		 * This can only happen if there is a driver that does not
		 * understand chaining. Here biodone will reissue the call
		 * to strategy for the incomplete buffers.
		 */
		printf("initiate_write_filepage: already started\n");
		return;
	}
	pagedep->pd_state |= IOSTARTED;
	for (i = 0; i < DAHASHSZ; i++) {
		LIST_FOREACH(dap, &pagedep->pd_diraddhd[i], da_pdlist) {
			ep = (struct direct *)
			    ((char *)bp->b_data + dap->da_offset);
			if (ep->d_ino != dap->da_newinum) {
				FREE_LOCK(&lk);
				panic("%s: dir inum %u != new %u",
				    "initiate_write_filepage",
				    ep->d_ino, dap->da_newinum);
			}
			if (dap->da_state & DIRCHG)
				ep->d_ino = dap->da_previous->dm_oldinum;
			else
				ep->d_ino = 0;
			dap->da_state &= ~ATTACHED;
			dap->da_state |= UNDONE;
		}
	}
}

/*
 * Called from within the procedure above to deal with unsatisfied
 * allocation dependencies in an inodeblock. The buffer must be
 * locked, thus, no I/O completion operations can occur while we
 * are manipulating its associated dependencies.
 */
/* The inode block */
STATIC void
initiate_write_inodeblock_ufs1(struct inodedep *inodedep, struct buf *bp)
{
	struct allocdirect *adp, *lastadp;
	struct ufs1_dinode *dp;
	struct fs *fs;
#ifdef DIAGNOSTIC
	daddr_t prevlbn = 0;
	int32_t d1, d2;
#endif
	int i, deplist;

	if (inodedep->id_state & IOSTARTED) {
		FREE_LOCK(&lk);
		panic("initiate_write_inodeblock: already started");
	}
	inodedep->id_state |= IOSTARTED;
	fs = inodedep->id_fs;
	dp = (struct ufs1_dinode *)bp->b_data +
	    ino_to_fsbo(fs, inodedep->id_ino);
	/*
	 * If the bitmap is not yet written, then the allocated
	 * inode cannot be written to disk.
	 */
	if ((inodedep->id_state & DEPCOMPLETE) == 0) {
		if (inodedep->id_savedino1 != NULL) {
			FREE_LOCK(&lk);
			panic("initiate_write_inodeblock: already doing I/O");
		}
		FREE_LOCK(&lk);
		inodedep->id_savedino1 = malloc(sizeof(struct ufs1_dinode),
		    M_INODEDEP, M_WAITOK);
		inodedep->id_unsize = sizeof(struct ufs1_dinode);
		ACQUIRE_LOCK(&lk);
		*inodedep->id_savedino1 = *dp;
		memset(dp, 0, sizeof(struct ufs1_dinode));
		return;
	}
	/*
	 * If no dependencies, then there is nothing to roll back.
	 */
	inodedep->id_savedsize = dp->di_size;
	if (TAILQ_FIRST(&inodedep->id_inoupdt) == NULL)
		return;
	/*
```

```c
         * Set the dependencies to busy.
         */
        for (deplist = 0, adp = TAILQ_FIRST(&inodedep->id_inoupdt); adp;
             adp = TAILQ_NEXT(adp, ad_next)) {
#ifdef DIAGNOSTIC
                if (deplist != 0 && prevlbn >= adp->ad_lbn) {
                        FREE_LOCK(&lk);
                        panic("softdep_write_inodeblock: lbn order");
                }
                prevlbn = adp->ad_lbn;
                if (adp->ad_lbn < NDADDR &&
                    (d1 = dp->di_db[adp->ad_lbn]) != (d2 = adp->ad_newblkno)) {
                        FREE_LOCK(&lk);
                        panic("%s: direct pointer #%lld mismatch %d != %d",
                            "softdep_write_inodeblock", (long long)adp->ad_lbn,
                            d1, d2);
                }
                if (adp->ad_lbn >= NDADDR &&
                    (d1 = dp->di_ib[adp->ad_lbn - NDADDR]) !=
                    (d2 = adp->ad_newblkno)) {
                        FREE_LOCK(&lk);
                        panic("%s: indirect pointer #%lld mismatch %d != %d",
                            "softdep_write_inodeblock", (long long)(adp->ad_lbn -
                            NDADDR), d1, d2);
                }
                deplist |= 1 << adp->ad_lbn;
                if ((adp->ad_state & ATTACHED) == 0) {
                        FREE_LOCK(&lk);
                        panic("softdep_write_inodeblock: Unknown state 0x%x",
                            adp->ad_state);
                }
#endif /* DIAGNOSTIC */
                adp->ad_state &= ~ATTACHED;
                adp->ad_state |= UNDONE;
        }
        /*
         * The on-disk inode cannot claim to be any larger than the last
         * fragment that has been written. Otherwise, the on-disk inode
         * might have fragments that were not the last block in the file
         * which would corrupt the filesystem.
         */
        for (lastadp = NULL, adp = TAILQ_FIRST(&inodedep->id_inoupdt); adp;
             lastadp = adp, adp = TAILQ_NEXT(adp, ad_next)) {
                if (adp->ad_lbn >= NDADDR)
                        break;
                dp->di_db[adp->ad_lbn] = adp->ad_oldblkno;
                /* keep going until hitting a rollback to a frag */
                if (adp->ad_oldsize == 0 || adp->ad_oldsize == fs->fs_bsize)
                        continue;
                dp->di_size = fs->fs_bsize * adp->ad_lbn + adp->ad_oldsize;
                for (i = adp->ad_lbn + 1; i < NDADDR; i++) {
#ifdef DIAGNOSTIC
                        if (dp->di_db[i] != 0 && (deplist & (1 << i)) == 0) {
                                FREE_LOCK(&lk);
                                panic("softdep_write_inodeblock: lost dep1");
                        }
#endif /* DIAGNOSTIC */
                        dp->di_db[i] = 0;
                }
                for (i = 0; i < NIADDR; i++) {
#ifdef DIAGNOSTIC
                        if (dp->di_ib[i] != 0 &&
                            (deplist & ((1 << NDADDR) << i)) == 0) {
                                FREE_LOCK(&lk);
                                panic("softdep_write_inodeblock: lost dep2");
                        }
#endif /* DIAGNOSTIC */
                        dp->di_ib[i] = 0;
                }
                return;
        }
        /*
         * If we have zero'ed out the last allocated block of the file,
         * roll back the size to the last currently allocated block.
         * We know that this last allocated block is a full-sized as
         * we already checked for fragments in the loop above.
         */
        if (lastadp != NULL &&
            dp->di_size <= (lastadp->ad_lbn + 1) * fs->fs_bsize) {
                for (i = lastadp->ad_lbn; i >= 0; i--)
                        if (dp->di_db[i] != 0)
                                break;
                dp->di_size = (i + 1) * fs->fs_bsize;
        }
        /*
         * The only dependencies are for indirect blocks.
         *
         * The file size for indirect block additions is not guaranteed.
         * Such a guarantee would be non-trivial to achieve. The conventional
         * synchronous write implementation also does not make this guarantee.
         * Fsck should catch and fix discrepancies. Arguably, the file size
         * can be over-estimated without destroying integrity when the file
         * moves into the indirect blocks (i.e., is large). If we want to
         * postpone fsck, we are stuck with this argument.
         */
        for (; adp; adp = TAILQ_NEXT(adp, ad_next))
                dp->di_ib[adp->ad_lbn - NDADDR] = 0;
}

#ifdef FFS2
/*
 * Version of initiate_write_inodeblock that handles FFS2 dinodes.
 */
/* The inode block */
STATIC void
initiate_write_inodeblock_ufs2(struct inodedep *inodedep, struct buf *bp)
{
        struct allocdirect *adp, *lastadp;
        struct ufs2_dinode *dp;
        struct fs *fs = inodedep->id_fs;
#ifdef DIAGNOSTIC
        daddr_t prevlbn = -1, d1, d2;
#endif
        int deplist, i;

        if (inodedep->id_state & IOSTARTED)
                panic("initiate_write_inodeblock_ufs2: already started");
        inodedep->id_state |= IOSTARTED;
        fs = inodedep->id_fs;
        dp = (struct ufs2_dinode *)bp->b_data +
            ino_to_fsbo(fs, inodedep->id_ino);
        /*
         * If the bitmap is not yet written, then the allocated
         * inode cannot be written to disk.
         */
        if ((inodedep->id_state & DEPCOMPLETE) == 0) {
                if (inodedep->id_savedino2 != NULL)
                        panic("initiate_write_inodeblock_ufs2: I/O underway");
                inodedep->id_savedino2 = malloc(sizeof(struct ufs2_dinode),
                    M_INODEDEP, M_WAITOK);
                inodedep->id_unsize = sizeof(struct ufs2_dinode);
                *inodedep->id_savedino2 = *dp;
                memset(dp, 0, sizeof(struct ufs2_dinode));
                return;
        }
        /*
         * If no dependencies, then there is nothing to roll back.
         */
        inodedep->id_savedsize = dp->di_size;
        if (TAILQ_FIRST(&inodedep->id_inoupdt) == NULL)
                return;

#ifdef notyet
        inodedep->id_savedextsize = dp->di_extsize;
        if (TAILQ_FIRST(&inodedep->id_inoupdt) == NULL &&
            TAILQ_FIRST(&inodedep->id_extupdt) == NULL)
                return;
        /*
         * Set the ext data dependencies to busy.
         */
        for (deplist = 0, adp = TAILQ_FIRST(&inodedep->id_extupdt); adp;
             adp = TAILQ_NEXT(adp, ad_next)) {
#ifdef DIAGNOSTIC
                if (deplist != 0 && prevlbn >= adp->ad_lbn) {
                        FREE_LOCK(&lk);
                        panic("softdep_write_inodeblock: lbn order");
                }
                prevlbn = adp->ad_lbn;
                if ((d1 = dp->di_extb[adp->ad_lbn]) !=
                    (d2 = adp->ad_newblkno)) {
                        FREE_LOCK(&lk);
                        panic("%s: direct pointer #%lld mismatch %lld != %lld",
                            "softdep_write_inodeblock", (long long)adp->ad_lbn,
                            d1, d2);
                }
                deplist |= 1 << adp->ad_lbn;
                if ((adp->ad_state & ATTACHED) == 0) {
                        FREE_LOCK(&lk);
                        panic("softdep_write_inodeblock: Unknown state 0x%x",
                            adp->ad_state);
                }
#endif /* DIAGNOSTIC */
                adp->ad_state &= ~ATTACHED;
                adp->ad_state |= UNDONE;
        }
        /*
         * The on-disk inode cannot claim to be any larger than the last
         * fragment that has been written. Otherwise, the on-disk inode
         * might have fragments that were not the last block in the ext
         * data which would corrupt the filesystem.
         */
        for (lastadp = NULL, adp = TAILQ_FIRST(&inodedep->id_extupdt); adp;
             lastadp = adp, adp = TAILQ_NEXT(adp, ad_next)) {
                dp->di_extb[adp->ad_lbn] = adp->ad_oldblkno;
                /* keep going until hitting a rollback to a frag */
                if (adp->ad_oldsize == 0 || adp->ad_oldsize == fs->fs_bsize)
                        continue;
                dp->di_extsize = fs->fs_bsize * adp->ad_lbn + adp->ad_oldsize;
                for (i = adp->ad_lbn + 1; i < NXADDR; i++) {
#ifdef DIAGNOSTIC
                        if (dp->di_extb[i] != 0 && (deplist & (1 << i)) == 0) {
                                FREE_LOCK(&lk);
                                panic("softdep_write_inodeblock: lost dep1");
                        }
#endif /* DIAGNOSTIC */
                        dp->di_extb[i] = 0;
                }
                lastadp = NULL;
                break;
        }
        /*
         * If we have zero'ed out the last allocated block of the ext
         * data, roll back the size to the last currently allocated block.
         * We know that this last allocated block is a full-sized as
         * we already checked for fragments in the loop above.
         */
        if (lastadp != NULL &&
            dp->di_extsize <= (lastadp->ad_lbn + 1) * fs->fs_bsize) {
                for (i = lastadp->ad_lbn; i >= 0; i--)
                        if (dp->di_extb[i] != 0)
                                break;
                dp->di_extsize = (i + 1) * fs->fs_bsize;
        }
#endif /* notyet */

        /*
         * Set the file data dependencies to busy.
         */
        for (deplist = 0, adp = TAILQ_FIRST(&inodedep->id_inoupdt); adp;
             adp = TAILQ_NEXT(adp, ad_next)) {
#ifdef DIAGNOSTIC
                if (deplist != 0 && prevlbn >= adp->ad_lbn) {
                        FREE_LOCK(&lk);
                        panic("softdep_write_inodeblock: lbn order");
                }
                prevlbn = adp->ad_lbn;
                if (adp->ad_lbn < NDADDR &&
                    (d1 = dp->di_db[adp->ad_lbn]) != (d2 = adp->ad_newblkno)) {
                        FREE_LOCK(&lk);
                        panic("%s: direct pointer #%lld mismatch %lld != %lld",
                            "softdep_write_inodeblock", (long long)adp->ad_lbn,
                            d1, d2);
                }
                if (adp->ad_lbn >= NDADDR &&
                    (d1 = dp->di_ib[adp->ad_lbn - NDADDR]) !=
                    (d2 = adp->ad_newblkno)) {
                        FREE_LOCK(&lk);
                        panic("%s: indirect pointer #%lld mismatch %lld != %lld",
                            "softdep_write_inodeblock", (long long)(adp->ad_lbn -
                            NDADDR), d1, d2);
                }
                deplist |= 1 << adp->ad_lbn;
                if ((adp->ad_state & ATTACHED) == 0) {
                        FREE_LOCK(&lk);
                        panic("softdep_write_inodeblock: Unknown state 0x%x",
                            adp->ad_state);
                }
#endif /* DIAGNOSTIC */
                adp->ad_state &= ~ATTACHED;
                adp->ad_state |= UNDONE;
        }
        /*
         * The on-disk inode cannot claim to be any larger than the last
         * fragment that has been written. Otherwise, the on-disk inode
         * might have fragments that were not the last block in the file
         * which would corrupt the filesystem.
         */
        for (lastadp = NULL, adp = TAILQ_FIRST(&inodedep->id_inoupdt); adp;
             lastadp = adp, adp = TAILQ_NEXT(adp, ad_next)) {
                if (adp->ad_lbn >= NDADDR)
                        break;
                dp->di_db[adp->ad_lbn] = adp->ad_oldblkno;
                /* keep going until hitting a rollback to a frag */
                if (adp->ad_oldsize == 0 || adp->ad_oldsize == fs->fs_bsize)
                        continue;
                dp->di_size = fs->fs_bsize * adp->ad_lbn + adp->ad_oldsize;
                for (i = adp->ad_lbn + 1; i < NDADDR; i++) {
#ifdef DIAGNOSTIC
                        if (dp->di_db[i] != 0 && (deplist & (1 << i)) == 0) {
                                FREE_LOCK(&lk);
                                panic("softdep_write_inodeblock: lost dep2");
                        }
#endif /* DIAGNOSTIC */
                        dp->di_db[i] = 0;
                }
                for (i = 0; i < NIADDR; i++) {
#ifdef DIAGNOSTIC
                        if (dp->di_ib[i] != 0 &&
                            (deplist & ((1 << NDADDR) << i)) == 0) {
                                FREE_LOCK(&lk);
                                panic("softdep_write_inodeblock: lost dep3");
                        }
#endif /* DIAGNOSTIC */
                        dp->di_ib[i] = 0;
                }
                return;
        }
        /*
         * If we have zero'ed out the last allocated block of the file,
         * roll back the size to the last currently allocated block.
         * We know that this last allocated block is a full-sized as
         * we already checked for fragments in the loop above.
         */
        if (lastadp != NULL &&
            dp->di_size <= (lastadp->ad_lbn + 1) * fs->fs_bsize) {
                for (i = lastadp->ad_lbn; i >= 0; i--)
                        if (dp->di_db[i] != 0)
                                break;
                dp->di_size = (i + 1) * fs->fs_bsize;
        }
        /*
         * The only dependencies are for indirect blocks.
         *
         * The file size for indirect block additions is not guaranteed.
         * Such a guarantee would be non-trivial to achieve. The conventional
         * synchronous write implementation also does not make this guarantee.
         * Fsck should catch and fix discrepancies. Arguably, the file size
         * can be over-estimated without destroying integrity when the file
         * moves into the indirect blocks (i.e., is large). If we want to
         * postpone fsck, we are stuck with this argument.
         */
        for (; adp; adp = TAILQ_NEXT(adp, ad_next))
                dp->di_ib[adp->ad_lbn - NDADDR] = 0;
}
#endif /* FFS2 */

/*
 * This routine is called during the completion interrupt
 * service routine for a disk write (from the procedure called
 * by the device driver to inform the file system caches of
 * a request completion).  It should be called early in this
 * procedure, before the block is made available to other
 * processes or other routines are called.
 */
/* describes the completed disk write */
void
softdep_disk_write_complete(struct buf *bp)
{
        struct worklist *wk;
        struct workhead reattach;
        struct newblk *newblk;
        struct allocindir *aip;
        struct allocdirect *adp;
        struct indirdep *indirdep;
        struct inodedep *inodedep;
        struct bmsafemap *bmsafemap;

        /*
         * If an error occurred while doing the write, then the data
         * has not hit the disk and the dependencies cannot be unrolled.
         */
        if ((bp->b_flags & B_ERROR) && !(bp->b_flags & B_INVAL))
                return;

#ifdef DEBUG
        if (lk.lkt_held != -1)
                panic("softdep_disk_write_complete: lock is held");
        lk.lkt_held = -2;
#endif
        LIST_INIT(&reattach);
        while ((wk = LIST_FIRST(&bp->b_dep)) != NULL) {
                WORKLIST_REMOVE(wk);
                switch (wk->wk_type) {

                case D_PAGEDEP:
                        if (handle_written_filepage(WK_PAGEDEP(wk), bp))
                                WORKLIST_INSERT(&reattach, wk);
                        continue;

                case D_INODEDEP:
                        if (handle_written_inodeblock(WK_INODEDEP(wk), bp))
                                WORKLIST_INSERT(&reattach, wk);
                        continue;

                case D_BMSAFEMAP:
                        bmsafemap = WK_BMSAFEMAP(wk);
                        while ((newblk = LIST_FIRST(&bmsafemap->sm_newblkhd))) {
                                newblk->nb_state |= DEPCOMPLETE;
                                newblk->nb_bmsafemap = NULL;
                                LIST_REMOVE(newblk, nb_deps);
                        }
                        while ((adp =
                            LIST_FIRST(&bmsafemap->sm_allocdirecthd))) {
                                adp->ad_state |= DEPCOMPLETE;
                                adp->ad_buf = NULL;
                                LIST_REMOVE(adp, ad_deps);
                                handle_allocdirect_partdone(adp);
                        }
                        while ((aip =
                            LIST_FIRST(&bmsafemap->sm_allocindirhd))) {
                                aip->ai_state |= DEPCOMPLETE;
                                aip->ai_buf = NULL;
                                LIST_REMOVE(aip, ai_deps);
                                handle_allocindir_partdone(aip);
                        }
                        while ((inodedep =
                             LIST_FIRST(&bmsafemap->sm_inodedephd)) != NULL) {
                                inodedep->id_state |= DEPCOMPLETE;
                                LIST_REMOVE(inodedep, id_deps);
                                inodedep->id_buf = NULL;
                        }
                        WORKITEM_FREE(bmsafemap, D_BMSAFEMAP);
                        continue;

                case D_MKDIR:
                        handle_written_mkdir(WK_MKDIR(wk), MKDIR_BODY);
                        continue;

                case D_ALLOCDIRECT:
                        adp = WK_ALLOCDIRECT(wk);
                        adp->ad_state |= COMPLETE;
                        handle_allocdirect_partdone(adp);
                        continue;

                case D_ALLOCINDIR:
                        aip = WK_ALLOCINDIR(wk);
                        aip->ai_state |= COMPLETE;
                        handle_allocindir_partdone(aip);
                        continue;

                case D_INDIRDEP:
                        indirdep = WK_INDIRDEP(wk);
                        if (indirdep->ir_state & GOINGAWAY)
                                panic("disk_write_complete: indirdep gone");
                        memcpy(bp->b_data, indirdep->ir_saveddata, bp->b_bcount);
                        free(indirdep->ir_saveddata, M_INDIRDEP, bp->b_bcount);
                        indirdep->ir_saveddata = NULL;
                        indirdep->ir_state &= ~UNDONE;
                        indirdep->ir_state |= ATTACHED;
                        while ((aip = LIST_FIRST(&indirdep->ir_donehd))) {
                                handle_allocindir_partdone(aip);
                                if (aip == LIST_FIRST(&indirdep->ir_donehd))
                                        panic("disk_write_complete: not gone");
                        }
                        WORKLIST_INSERT(&reattach, wk);
                        if ((bp->b_flags & B_DELWRI) == 0)
                                stat_indir_blk_ptrs++;
                        buf_dirty(bp);
                        continue;

                default:
                        panic("handle_disk_write_complete: Unknown type %s",
                            TYPENAME(wk->wk_type));
                        /* NOTREACHED */
                }
        }
        /*
         * Reattach any requests that must be redone.
         */
        while ((wk = LIST_FIRST(&reattach)) != NULL) {
                WORKLIST_REMOVE(wk);
                WORKLIST_INSERT(&bp->b_dep, wk);
        }
#ifdef DEBUG
        if (lk.lkt_held != -2)
                panic("softdep_disk_write_complete: lock lost");
        lk.lkt_held = -1;
#endif
}

/*
 * Called from within softdep_disk_write_complete above. Note that
 * this routine is always called from interrupt level with further
 * splbio interrupts blocked.
 */
/* the completed allocdirect */
STATIC void
handle_allocdirect_partdone(struct allocdirect *adp)
{
        struct allocdirect *listadp;
        struct inodedep *inodedep;
        long bsize, delay;

        splassert(IPL_BIO);

        if ((adp->ad_state & ALLCOMPLETE) != ALLCOMPLETE)
                return;
        if (adp->ad_buf != NULL)
                panic("handle_allocdirect_partdone: dangling dep");

        /*
         * The on-disk inode cannot claim to be any larger than the last
         * fragment that has been written. Otherwise, the on-disk inode
         * might have fragments that were not the last block in the file
         * which would corrupt the filesystem. Thus, we cannot free any
         * allocdirects after one whose ad_oldblkno claims a fragment as
         * these blocks must be rolled back to zero before writing the inode.
         * We check the currently active set of allocdirects in id_inoupdt.
         */
        inodedep = adp->ad_inodedep;
        bsize = inodedep->id_fs->fs_bsize;
        TAILQ_FOREACH(listadp, &inodedep->id_inoupdt, ad_next) {
                /* found our block */
                if (listadp == adp)
                        break;
                /* continue if ad_oldlbn is not a fragment */
                if (listadp->ad_oldsize == 0 ||
                    listadp->ad_oldsize == bsize)
                        continue;
                /* hit a fragment */
                return;
        }
        /*
         * If we have reached the end of the current list without
         * finding the just finished dependency, then it must be
         * on the future dependency list. Future dependencies cannot
         * be freed until they are moved to the current list.
         */
        if (listadp == NULL) {
#ifdef DEBUG
                TAILQ_FOREACH(listadp, &inodedep->id_newinoupdt, ad_next)
                        /* found our block */
                        if (listadp == adp)
                                break;
                if (listadp == NULL)
                        panic("handle_allocdirect_partdone: lost dep");
#endif /* DEBUG */
                return;
        }
        /*
         * If we have found the just finished dependency, then free
         * it along with anything that follows it that is complete.
         * If the inode still has a bitmap dependency, then it has
         * never been written to disk, hence the on-disk inode cannot
         * reference the old fragment so we can free it without delay.
         */
        delay = (inodedep->id_state & DEPCOMPLETE);
        for (; adp; adp = listadp) {
                listadp = TAILQ_NEXT(adp, ad_next);
                if ((adp->ad_state & ALLCOMPLETE) != ALLCOMPLETE)
                        return;
                free_allocdirect(&inodedep->id_inoupdt, adp, delay);
        }
}

/*
 * Called from within softdep_disk_write_complete above. Note that
 * this routine is always called from interrupt level with further
 * splbio interrupts blocked.
 */
/* the completed allocindir */
STATIC void
handle_allocindir_partdone(struct allocindir *aip)
{
        struct indirdep *indirdep;

        splassert(IPL_BIO);

        if ((aip->ai_state & ALLCOMPLETE) != ALLCOMPLETE)
                return;
        if (aip->ai_buf != NULL)
                panic("handle_allocindir_partdone: dangling dependency");
        indirdep = aip->ai_indirdep;
        if (indirdep->ir_state & UNDONE) {
                LIST_REMOVE(aip, ai_next);
                LIST_INSERT_HEAD(&indirdep->ir_donehd, aip, ai_next);
                return;
        }
        if (indirdep->ir_state & UFS1FMT)
                ((int32_t *)indirdep->ir_savebp->b_data)[aip->ai_offset] =
                    aip->ai_newblkno;
        else
                ((int64_t *)indirdep->ir_savebp->b_data)[aip->ai_offset] =
                    aip->ai_newblkno;
        LIST_REMOVE(aip, ai_next);
        if (aip->ai_freefrag != NULL)
                add_to_worklist(&aip->ai_freefrag->ff_list);
        WORKITEM_FREE(aip, D_ALLOCINDIR);
}

/*
 * Called from within softdep_disk_write_complete above to restore
 * in-memory inode block contents to their most up-to-date state. Note
 * that this routine is always called from interrupt level with further
 * splbio interrupts blocked.
 */
/* buffer containing the inode block */
STATIC int
handle_written_inodeblock(struct inodedep *inodedep, struct buf *bp)
{
        struct worklist *wk, *filefree;
        struct allocdirect *adp, *nextadp;
        struct ufs1_dinode *dp1 = NULL;
        struct ufs2_dinode *dp2 = NULL;
        int hadchanges, fstype;

        splassert(IPL_BIO);

        if ((inodedep->id_state & IOSTARTED) == 0)
                panic("handle_written_inodeblock: not started");
        inodedep->id_state &= ~IOSTARTED;

        if (inodedep->id_fs->fs_magic == FS_UFS1_MAGIC) {
                fstype = UM_UFS1;
                dp1 = (struct ufs1_dinode *) bp->b_data +
                    ino_to_fsbo(inodedep->id_fs, inodedep->id_ino);
        } else {
                fstype = UM_UFS2;
                dp2 = (struct ufs2_dinode *) bp->b_data +
                    ino_to_fsbo(inodedep->id_fs, inodedep->id_ino);
        }

        /*
         * If we had to rollback the inode allocation because of
         * bitmaps being incomplete, then simply restore it.
         * Keep the block dirty so that it will not be reclaimed until
         * all associated dependencies have been cleared and the
         * corresponding updates written to disk.
         */
        if (inodedep->id_savedino1 != NULL) {
                if (fstype == UM_UFS1)
                        *dp1 = *inodedep->id_savedino1;
                else
                        *dp2 = *inodedep->id_savedino2;
                free(inodedep->id_savedino1, M_INODEDEP, inodedep->id_unsize);
                inodedep->id_savedino1 = NULL;
                if ((bp->b_flags & B_DELWRI) == 0)
                        stat_inode_bitmap++;
                buf_dirty(bp);
                return (1);
        }
        inodedep->id_state |= COMPLETE;
        /*
         * Roll forward anything that had to be rolled back before
         * the inode could be updated.
         */
        hadchanges = 0;
        for (adp = TAILQ_FIRST(&inodedep->id_inoupdt); adp; adp = nextadp) {
                nextadp = TAILQ_NEXT(adp, ad_next);
                if (adp->ad_state & ATTACHED)
                        panic("handle_written_inodeblock: new entry");
                if (fstype == UM_UFS1) {
                        if (adp->ad_lbn < NDADDR) {
                                if (dp1->di_db[adp->ad_lbn] != adp->ad_oldblkno)
                                         panic("%s: %s #%lld mismatch %d != "
                                             "%lld",
                                             "handle_written_inodeblock",
                                             "direct pointer",
                                             (long long)adp->ad_lbn,
                                             dp1->di_db[adp->ad_lbn],
                                             (long long)adp->ad_oldblkno);
                                dp1->di_db[adp->ad_lbn] = adp->ad_newblkno;
                        } else {
                                if (dp1->di_ib[adp->ad_lbn - NDADDR] != 0)
                                        panic("%s: %s #%lld allocated as %d",
                                            "handle_written_inodeblock",
                                            "indirect pointer",
                                            (long long)(adp->ad_lbn - NDADDR),
                                            dp1->di_ib[adp->ad_lbn - NDADDR]);
                                dp1->di_ib[adp->ad_lbn - NDADDR] =
                                   adp->ad_newblkno;
                        }
                } else {
                        if (adp->ad_lbn < NDADDR) {
                                if (dp2->di_db[adp->ad_lbn] != adp->ad_oldblkno)
                                        panic("%s: %s #%lld mismatch %lld != "
                                            "%lld", "handle_written_inodeblock",
                                            "direct pointer",
                                            (long long)adp->ad_lbn,
                                            dp2->di_db[adp->ad_lbn],
                                            (long long)adp->ad_oldblkno);
                                dp2->di_db[adp->ad_lbn] = adp->ad_newblkno;
                        } else {
                                if (dp2->di_ib[adp->ad_lbn - NDADDR] != 0)
                                        panic("%s: %s #%lld allocated as %lld",
                                            "handle_written_inodeblock",
                                            "indirect pointer",
                                            (long long)(adp->ad_lbn - NDADDR),
                                            dp2->di_ib[adp->ad_lbn - NDADDR]);
                                dp2->di_ib[adp->ad_lbn - NDADDR] =
                                    adp->ad_newblkno;
                        }
                }
                adp->ad_state &= ~UNDONE;
                adp->ad_state |= ATTACHED;
                hadchanges = 1;
        }
        if (hadchanges && (bp->b_flags & B_DELWRI) == 0)
                stat_direct_blk_ptrs++;
        /*
         * Reset the file size to its most up-to-date value.
         */
        if (inodedep->id_savedsize == -1)
                panic("handle_written_inodeblock: bad size");

        if (fstype == UM_UFS1) {
                if (dp1->di_size != inodedep->id_savedsize) {
                        dp1->di_size = inodedep->id_savedsize;
                        hadchanges = 1;
                }
        } else {
                if (dp2->di_size != inodedep->id_savedsize) {
                        dp2->di_size = inodedep->id_savedsize;
                        hadchanges = 1;
                }
        }
        inodedep->id_savedsize = -1;
        /*
         * If there were any rollbacks in the inode block, then it must be
         * marked dirty so that its will eventually get written back in
         * its correct form.
         */
        if (hadchanges)
                buf_dirty(bp);
        /*
         * Process any allocdirects that completed during the update.
         */
        if ((adp = TAILQ_FIRST(&inodedep->id_inoupdt)) != NULL)
                handle_allocdirect_partdone(adp);
        /*
         * Process deallocations that were held pending until the
         * inode had been written to disk. Freeing of the inode
         * is delayed until after all blocks have been freed to
         * avoid creation of new <vfsid, inum, lbn> triples
         * before the old ones have been deleted.
         */
        filefree = NULL;
        while ((wk = LIST_FIRST(&inodedep->id_bufwait)) != NULL) {
                WORKLIST_REMOVE(wk);
                switch (wk->wk_type) {

                case D_FREEFILE:
                        /*
                         * We defer adding filefree to the worklist until
                         * all other additions have been made to ensure
                         * that it will be done after all the old blocks
                         * have been freed.
                         */
                        if (filefree != NULL)
                                panic("handle_written_inodeblock: filefree");
                        filefree = wk;
                        continue;

                case D_MKDIR:
                        handle_written_mkdir(WK_MKDIR(wk), MKDIR_PARENT);
                        continue;

                case D_DIRADD:
                        diradd_inode_written(WK_DIRADD(wk), inodedep);
                        continue;

                case D_FREEBLKS:
                        wk->wk_state |= COMPLETE;
                        if ((wk->wk_state & ALLCOMPLETE) != ALLCOMPLETE)
                                continue;
                        /* FALLTHROUGH */
                case D_FREEFRAG:
                case D_DIRREM:
                        add_to_worklist(wk);
                        continue;

                case D_NEWDIRBLK:
                        free_newdirblk(WK_NEWDIRBLK(wk));
                        continue;

                default:
                        panic("handle_written_inodeblock: Unknown type %s",
                            TYPENAME(wk->wk_type));
                        /* NOTREACHED */
                }
        }
        if (filefree != NULL) {
                if (free_inodedep(inodedep) == 0)
                        panic("handle_written_inodeblock: live inodedep");
                add_to_worklist(filefree);
                return (0);
        }

        /*
         * If no outstanding dependencies, free it.
         */
        if (free_inodedep(inodedep) ||
            TAILQ_FIRST(&inodedep->id_inoupdt) == NULL)
                return (0);
        return (hadchanges);
}

/*
 * Process a diradd entry after its dependent inode has been written.
 * This routine must be called with splbio interrupts blocked.
 */
STATIC void
diradd_inode_written(struct diradd *dap, struct inodedep *inodedep)
{
        struct pagedep *pagedep;

        splassert(IPL_BIO);

        dap->da_state |= COMPLETE;
        if ((dap->da_state & ALLCOMPLETE) == ALLCOMPLETE) {
                if (dap->da_state & DIRCHG)
                        pagedep = dap->da_previous->dm_pagedep;
                else
                        pagedep = dap->da_pagedep;
                LIST_REMOVE(dap, da_pdlist);
                LIST_INSERT_HEAD(&pagedep->pd_pendinghd, dap, da_pdlist);
        }
        WORKLIST_INSERT(&inodedep->id_pendinghd, &dap->da_list);
}

/*
 * Handle the completion of a mkdir dependency.
 */
STATIC void
handle_written_mkdir(struct mkdir *mkdir, int type)
{
        struct diradd *dap;
        struct pagedep *pagedep;

        splassert(IPL_BIO);

        if (mkdir->md_state != type)
                panic("handle_written_mkdir: bad type");
        dap = mkdir->md_diradd;
        dap->da_state &= ~type;
        if ((dap->da_state & (MKDIR_PARENT | MKDIR_BODY)) == 0)
                dap->da_state |= DEPCOMPLETE;
        if ((dap->da_state & ALLCOMPLETE) == ALLCOMPLETE) {
                if (dap->da_state & DIRCHG)
                        pagedep = dap->da_previous->dm_pagedep;
                else
                        pagedep = dap->da_pagedep;
                LIST_REMOVE(dap, da_pdlist);
                LIST_INSERT_HEAD(&pagedep->pd_pendinghd, dap, da_pdlist);
        }
        LIST_REMOVE(mkdir, md_mkdirs);
        WORKITEM_FREE(mkdir, D_MKDIR);
}

/*
 * Called from within softdep_disk_write_complete above.
 * A write operation was just completed. Removed inodes can
 * now be freed and associated block pointers may be committed.
 * Note that this routine is always called from interrupt level
 * with further splbio interrupts blocked.
 */
/* buffer containing the written page */
STATIC int
handle_written_filepage(struct pagedep *pagedep, struct buf *bp)
{
        struct dirrem *dirrem;
        struct diradd *dap, *nextdap;
        struct direct *ep;
        int i, chgs;

        splassert(IPL_BIO);

        if ((pagedep->pd_state & IOSTARTED) == 0)
                panic("handle_written_filepage: not started");
        pagedep->pd_state &= ~IOSTARTED;
        /*
         * Process any directory removals that have been committed.
         */
        while ((dirrem = LIST_FIRST(&pagedep->pd_dirremhd)) != NULL) {
                LIST_REMOVE(dirrem, dm_next);
                dirrem->dm_dirinum = pagedep->pd_ino;
                add_to_worklist(&dirrem->dm_list);
        }
        /*
         * Free any directory additions that have been committed.
         * If it is a newly allocated block, we have to wait until
         * the on-disk directory inode claims the new block.
         */
        if ((pagedep->pd_state & NEWBLOCK) == 0)
                while ((dap = LIST_FIRST(&pagedep->pd_pendinghd)) != NULL)
                        free_diradd(dap);
        /*
         * Uncommitted directory entries must be restored.
         */
        for (chgs = 0, i = 0; i < DAHASHSZ; i++) {
                for (dap = LIST_FIRST(&pagedep->pd_diraddhd[i]); dap;
                     dap = nextdap) {
                        nextdap = LIST_NEXT(dap, da_pdlist);
                        if (dap->da_state & ATTACHED)
                                panic("handle_written_filepage: attached");
                        ep = (struct direct *)
                            ((char *)bp->b_data + dap->da_offset);
                        ep->d_ino = dap->da_newinum;
                        dap->da_state &= ~UNDONE;
                        dap->da_state |= ATTACHED;
                        chgs = 1;
                        /*
                         * If the inode referenced by the directory has
                         * been written out, then the dependency can be
                         * moved to the pending list.
                         */
                        if ((dap->da_state & ALLCOMPLETE) == ALLCOMPLETE) {
                                LIST_REMOVE(dap, da_pdlist);
                                LIST_INSERT_HEAD(&pagedep->pd_pendinghd, dap,
                                    da_pdlist);
                        }
                }
        }
        /*
         * If there were any rollbacks in the directory, then it must be
         * marked dirty so that its will eventually get written back in
         * its correct form.
         */
        if (chgs) {
                if ((bp->b_flags & B_DELWRI) == 0)
                        stat_dir_entry++;
                buf_dirty(bp);
                return (1);
        }
        /*
         * If we are not waiting for a new directory block to be
         * claimed by its inode, then the pagedep will be freed.
         * Otherwise it will remain to track any new entries on
         * the page in case they are fsync'ed.
         */
        if ((pagedep->pd_state & NEWBLOCK) == 0) {
                LIST_REMOVE(pagedep, pd_hash);
                WORKITEM_FREE(pagedep, D_PAGEDEP);
        }
        return (0);
}

/*
 * Writing back in-core inode structures.
 *
 * The file system only accesses an inode's contents when it occupies an
 * "in-core" inode structure.  These "in-core" structures are separate from
 * the page frames used to cache inode blocks.  Only the latter are
 * transferred to/from the disk.  So, when the updated contents of the
 * "in-core" inode structure are copied to the corresponding in-memory inode
 * block, the dependencies are also transferred.  The following procedure is
 * called when copying a dirty "in-core" inode to a cached inode block.
 */

/*
 * Called when an inode is loaded from disk. If the effective link count
 * differed from the actual link count when it was last flushed, then we
 * need to ensure that the correct effective link count is put back.
 */
/* the "in_core" copy of the inode */
void
softdep_load_inodeblock(struct inode *ip)
{
	struct inodedep *inodedep;

	/*
	 * Check for alternate nlink count.
	 */
	ip->i_effnlink = DIP(ip, nlink);
	ACQUIRE_LOCK(&lk);
	if (inodedep_lookup(ip->i_fs, ip->i_number, 0, &inodedep) == 0) {
		FREE_LOCK(&lk);
		return;
	}
	ip->i_effnlink -= inodedep->id_nlinkdelta;
	FREE_LOCK(&lk);
}

/*
 * This routine is called just before the "in-core" inode
 * information is to be copied to the in-memory inode block.
 * Recall that an inode block contains several inodes. If
 * the force flag is set, then the dependencies will be
 * cleared so that the update can always be made. Note that
 * the buffer is locked when this routine is called, so we
 * will never be in the middle of writing the inode block
 * to disk.
 */
/* the "in_core" copy of the inode */
/* the buffer containing the inode block */
/* nonzero => update must be allowed */
void
softdep_update_inodeblock(struct inode *ip, struct buf *bp, int waitfor)
{
	struct inodedep *inodedep;
	struct worklist *wk;
	int error, gotit;

	/*
	 * If the effective link count is not equal to the actual link
	 * count, then we must track the difference in an inodedep while
	 * the inode is (potentially) tossed out of the cache. Otherwise,
	 * if there is no existing inodedep, then there are no dependencies
	 * to track.
	 */
	ACQUIRE_LOCK(&lk);
	if (inodedep_lookup(ip->i_fs, ip->i_number, 0, &inodedep) == 0) {
		FREE_LOCK(&lk);
		if (ip->i_effnlink != DIP(ip, nlink))
			panic("softdep_update_inodeblock: bad link count");
		return;
	}
	if (inodedep->id_nlinkdelta != DIP(ip, nlink) - ip->i_effnlink) {
		FREE_LOCK(&lk);
		panic("softdep_update_inodeblock: bad delta");
	}
	/*
	 * Changes have been initiated. Anything depending on these
	 * changes cannot occur until this inode has been written.
	 */
	inodedep->id_state &= ~COMPLETE;
	if ((inodedep->id_state & ONWORKLIST) == 0)
		WORKLIST_INSERT(&bp->b_dep, &inodedep->id_list);
	/*
	 * Any new dependencies associated with the incore inode must
	 * now be moved to the list associated with the buffer holding
	 * the in-memory copy of the inode. Once merged process any
	 * allocdirects that are completed by the merger.
	 */
	merge_inode_lists(inodedep);
	if (TAILQ_FIRST(&inodedep->id_inoupdt) != NULL)
		handle_allocdirect_partdone(TAILQ_FIRST(&inodedep->id_inoupdt));
	/*
	 * Now that the inode has been pushed into the buffer, the
	 * operations dependent on the inode being written to disk
	 * can be moved to the id_bufwait so that they will be
	 * processed when the buffer I/O completes.
	 */
	while ((wk = LIST_FIRST(&inodedep->id_inowait)) != NULL) {
		WORKLIST_REMOVE(wk);
		WORKLIST_INSERT(&inodedep->id_bufwait, wk);
	}
	/*
	 * Newly allocated inodes cannot be written until the bitmap
	 * that allocates them have been written (indicated by
	 * DEPCOMPLETE being set in id_state). If we are doing a
	 * forced sync (e.g., an fsync on a file), we force the bitmap
	 * to be written so that the update can be done.
	 */
	do {
		if ((inodedep->id_state & DEPCOMPLETE) != 0 || waitfor == 0) {
			FREE_LOCK(&lk);
			return;
		}
		bp = inodedep->id_buf;
		gotit = getdirtybuf(bp, MNT_WAIT);
	} while (gotit == -1);
	FREE_LOCK(&lk);
	if (gotit && (error = bwrite(bp)) != 0)
		softdep_error("softdep_update_inodeblock: bwrite", error);
	if ((inodedep->id_state & DEPCOMPLETE) == 0)
		panic("softdep_update_inodeblock: update failed");
}

/*
 * Merge the new inode dependency list (id_newinoupdt) into the old
 * inode dependency list (id_inoupdt). This routine must be called
 * with splbio interrupts blocked.
 */
STATIC void
merge_inode_lists(struct inodedep *inodedep)
{
	struct allocdirect *listadp, *newadp;

	splassert(IPL_BIO);

	newadp = TAILQ_FIRST(&inodedep->id_newinoupdt);
	for (listadp = TAILQ_FIRST(&inodedep->id_inoupdt); listadp && newadp;) {
		if (listadp->ad_lbn < newadp->ad_lbn) {
			listadp = TAILQ_NEXT(listadp, ad_next);
			continue;
		}
		TAILQ_REMOVE(&inodedep->id_newinoupdt, newadp, ad_next);
		TAILQ_INSERT_BEFORE(listadp, newadp, ad_next);
		if (listadp->ad_lbn == newadp->ad_lbn) {
			allocdirect_merge(&inodedep->id_inoupdt, newadp,
			    listadp);
			listadp = newadp;
		}
		newadp = TAILQ_FIRST(&inodedep->id_newinoupdt);
	}
	TAILQ_CONCAT(&inodedep->id_inoupdt, &inodedep->id_newinoupdt, ad_next);
}

/*
 * If we are doing an fsync, then we must ensure that any directory
 * entries for the inode have been written after the inode gets to disk.
 */
/* the "in_core" copy of the inode */
int
softdep_fsync(struct vnode *vp)
{
	struct inodedep *inodedep;
	struct pagedep *pagedep;
	struct worklist *wk;
	struct diradd *dap;
	struct mount *mnt;
	struct vnode *pvp;
	struct inode *ip;
	struct inode *pip;
	struct buf *bp;
	struct fs *fs;
	struct proc *p = CURPROC;		/* XXX */
	int error, flushparent;
	ufsino_t parentino;
	daddr_t lbn;

	ip = VTOI(vp);
	fs = ip->i_fs;
	ACQUIRE_LOCK(&lk);
	if (inodedep_lookup(fs, ip->i_number, 0, &inodedep) == 0) {
		FREE_LOCK(&lk);
		return (0);
	}
	if (LIST_FIRST(&inodedep->id_inowait) != NULL ||
	    LIST_FIRST(&inodedep->id_bufwait) != NULL ||
	    TAILQ_FIRST(&inodedep->id_inoupdt) != NULL ||
	    TAILQ_FIRST(&inodedep->id_newinoupdt) != NULL) {
		FREE_LOCK(&lk);
		panic("softdep_fsync: pending ops");
	}
	for (error = 0, flushparent = 0; ; ) {
		if ((wk = LIST_FIRST(&inodedep->id_pendinghd)) == NULL)
			break;
		if (wk->wk_type != D_DIRADD) {
			FREE_LOCK(&lk);
			panic("softdep_fsync: Unexpected type %s",
			    TYPENAME(wk->wk_type));
		}
		dap = WK_DIRADD(wk);
		/*
		 * Flush our parent if this directory entry has a MKDIR_PARENT
		 * dependency or is contained in a newly allocated block.
		 */
		if (dap->da_state & DIRCHG)
			pagedep = dap->da_previous->dm_pagedep;
		else
			pagedep = dap->da_pagedep;
		mnt = pagedep->pd_mnt;
		parentino = pagedep->pd_ino;
		lbn = pagedep->pd_lbn;
		if ((dap->da_state & (MKDIR_BODY | COMPLETE)) != COMPLETE) {
			FREE_LOCK(&lk);
			panic("softdep_fsync: dirty");
		}
		if ((dap->da_state & MKDIR_PARENT) ||
		    (pagedep->pd_state & NEWBLOCK))
			flushparent = 1;
		else
			flushparent = 0;
		/*
		 * If we are being fsync'ed as part of vgone'ing this vnode,
		 * then we will not be able to release and recover the
		 * vnode below, so we just have to give up on writing its
		 * directory entry out. It will eventually be written, just
		 * not now, but then the user was not asking to have it
		 * written, so we are not breaking any promises.
		 */
		if (vp->v_flag & VXLOCK)
			break;
		/*
		 * We prevent deadlock by always fetching inodes from the
		 * root, moving down the directory tree. Thus, when fetching
		 * our parent directory, we must unlock ourselves before
		 * requesting the lock on our parent. See the comment in
		 * ufs_lookup for details on possible races.
		 */
		FREE_LOCK(&lk);
		VOP_UNLOCK(vp);
		error = VFS_VGET(mnt, parentino, &pvp);
		vn_lock(vp, LK_EXCLUSIVE | LK_RETRY);
		if (error != 0)
			return (error);
		/*
		 * All MKDIR_PARENT dependencies and all the NEWBLOCK pagedeps
		 * that are contained in direct blocks will be resolved by
		 * doing a UFS_UPDATE. Pagedeps contained in indirect blocks
		 * may require a complete sync'ing of the directory. So, we
		 * try the cheap and fast UFS_UPDATE first, and if that fails,
		 * then we do the slower VOP_FSYNC of the directory.
		 */
		pip = VTOI(pvp);
		if (flushparent) {
			error = UFS_UPDATE(pip, 1);
			if (error) {
				vput(pvp);
				return (error);
			}
			if (pagedep->pd_state & NEWBLOCK) {
				error = VOP_FSYNC(pvp, p->p_ucred, MNT_WAIT, p);
				if (error) {
					vput(pvp);
					return (error);
				}
			}
		}
		/*
		 * Flush directory page containing the inode's name.
		 */
		error = bread(pvp, lbn, fs->fs_bsize, &bp);
		if (error == 0) {
			bp->b_bcount = blksize(fs, pip, lbn);
			error = bwrite(bp);
		} else
			brelse(bp);
		vput(pvp);
		if (error != 0)
			return (error);
		ACQUIRE_LOCK(&lk);
		if (inodedep_lookup(fs, ip->i_number, 0, &inodedep) == 0)
			break;
	}
	FREE_LOCK(&lk);
	return (0);
}

/*
 * Flush all the dirty bitmaps associated with the block device
 * before flushing the rest of the dirty blocks so as to reduce
 * the number of dependencies that will have to be rolled back.
 */
void
softdep_fsync_mountdev(struct vnode *vp, int waitfor)
{
	struct buf *bp, *nbp;
	struct worklist *wk;

	if (!vn_isdisk(vp, NULL))
		panic("softdep_fsync_mountdev: vnode not a disk");
	ACQUIRE_LOCK(&lk);
	LIST_FOREACH_SAFE(bp, &vp->v_dirtyblkhd, b_vnbufs, nbp) {
		/*
		 * If it is already scheduled, skip to the next buffer.
		 */
		splassert(IPL_BIO);
		if (bp->b_flags & B_BUSY)
			continue;

		if ((bp->b_flags & B_DELWRI) == 0) {
			FREE_LOCK(&lk);
			panic("softdep_fsync_mountdev: not dirty");
```

```c
                }
                /*
                 * We are only interested in bitmaps with outstanding
                 * dependencies.
                 */
                if ((wk = LIST_FIRST(&bp->b_dep)) == NULL ||
                    wk->wk_type != D_BMSAFEMAP) {
                        continue;
                }
                bremfree(bp);
                buf_acquire(bp);
                FREE_LOCK(&lk);
                (void) bawrite(bp);
                ACQUIRE_LOCK(&lk);
                /*
                 * Since we may have slept during the I/O, we need
                 * to start from a known point.
                 */
                nbp = LIST_FIRST(&vp->v_dirtyblkhd);
        }
        if (waitfor == MNT_WAIT)
                drain_output(vp, 1);
        FREE_LOCK(&lk);
}

/*
 * This routine is called when we are trying to synchronously flush a
 * file. This routine must eliminate any filesystem metadata dependencies
 * so that the syncing routine can succeed by pushing the dirty blocks
 * associated with the file. If any I/O errors occur, they are returned.
 */
int
softdep_sync_metadata(struct vop_fsync_args *ap)
{
        struct vnode *vp = ap->a_vp;
        struct pagedep *pagedep;
        struct allocdirect *adp;
        struct allocindir *aip;
        struct buf *bp, *nbp;
        struct worklist *wk;
        int i, gotit, error, waitfor;

        /*
         * Check whether this vnode is involved in a filesystem
         * that is doing soft dependency processing.
         */
        if (!vn_isdisk(vp, NULL)) {
                if (!DOINGSOFTDEP(vp))
                        return (0);
        } else
                if (vp->v_specmountpoint == NULL ||
                    (vp->v_specmountpoint->mnt_flag & MNT_SOFTDEP) == 0)
                        return (0);
        /*
         * Ensure that any direct block dependencies have been cleared.
         */
        ACQUIRE_LOCK(&lk);
        if ((error = flush_inodedep_deps(VTOI(vp)->i_fs, VTOI(vp)->i_number))) {
                FREE_LOCK(&lk);
                return (error);
        }
        /*
         * For most files, the only metadata dependencies are the
         * cylinder group maps that allocate their inode or blocks.
         * The block allocation dependencies can be found by traversing
         * the dependency lists for any buffers that remain on their
         * dirty buffer list. The inode allocation dependency will
         * be resolved when the inode is updated with MNT_WAIT.
         * This work is done in two passes. The first pass grabs most
         * of the buffers and begins asynchronously writing them. The
         * only way to wait for these asynchronous writes is to sleep
         * on the filesystem vnode which may stay busy for a long time
         * if the filesystem is active. So, instead, we make a second
         * pass over the dependencies blocking on each write. In the
         * usual case we will be blocking against a write that we
         * initiated, so when it is done the dependency will have been
         * resolved. Thus the second pass is expected to end quickly.
         */
        waitfor = MNT_NOWAIT;
top:
        /*
         * We must wait for any I/O in progress to finish so that
         * all potential buffers on the dirty list will be visible.
         */
        drain_output(vp, 1);
        bp = LIST_FIRST(&vp->v_dirtyblkhd);
        gotit = getdirtybuf(bp, MNT_WAIT);
        if (gotit == 0) {
                FREE_LOCK(&lk);
                return (0);
        } else if (gotit == -1)
                goto top;
loop:
        /*
         * As we hold the buffer locked, none of its dependencies
         * will disappear.
         */
        LIST_FOREACH(wk, &bp->b_dep, wk_list) {
                switch (wk->wk_type) {

                case D_ALLOCDIRECT:
                        adp = WK_ALLOCDIRECT(wk);
                        if (adp->ad_state & DEPCOMPLETE)
                                break;
                        nbp = adp->ad_buf;
                        gotit = getdirtybuf(nbp, waitfor);
                        if (gotit == 0)
                                break;
                        else if (gotit == -1)
                                goto loop;
                        FREE_LOCK(&lk);
                        if (waitfor == MNT_NOWAIT) {
                                bawrite(nbp);
                        } else if ((error = VOP_BWRITE(nbp)) != 0) {
                                bawrite(bp);
                                return (error);
                        }
                        ACQUIRE_LOCK(&lk);
                        break;

                case D_ALLOCINDIR:
                        aip = WK_ALLOCINDIR(wk);
                        if (aip->ai_state & DEPCOMPLETE)
                                break;
                        nbp = aip->ai_buf;
                        gotit = getdirtybuf(nbp, waitfor);
                        if (gotit == 0)
                                break;
                        else if (gotit == -1)
                                goto loop;
                        FREE_LOCK(&lk);
                        if (waitfor == MNT_NOWAIT) {
                                bawrite(nbp);
                        } else if ((error = VOP_BWRITE(nbp)) != 0) {
                                bawrite(bp);
                                return (error);
                        }
                        ACQUIRE_LOCK(&lk);
                        break;

                case D_INDIRDEP:
                restart:

                        LIST_FOREACH(aip, &WK_INDIRDEP(wk)->ir_deplisthd, ai_next) {
                                if (aip->ai_state & DEPCOMPLETE)
                                        continue;
                                nbp = aip->ai_buf;
                                if (getdirtybuf(nbp, MNT_WAIT) <= 0)
                                        goto restart;
                                FREE_LOCK(&lk);
                                if ((error = VOP_BWRITE(nbp)) != 0) {
                                        bawrite(bp);
                                        return (error);
                                }
                                ACQUIRE_LOCK(&lk);
                                goto restart;
                        }
                        break;

                case D_INODEDEP:
                        if ((error = flush_inodedep_deps(WK_INODEDEP(wk)->id_fs,
                            WK_INODEDEP(wk)->id_ino)) != 0) {
                                FREE_LOCK(&lk);
                                bawrite(bp);
                                return (error);
                        }
                        break;

                case D_PAGEDEP:
                        /*
                         * We are trying to sync a directory that may
                         * have dependencies on both its own metadata
                         * and/or dependencies on the inodes of any
                         * recently allocated files. We walk its diradd
                         * lists pushing out the associated inode.
                         */
                        pagedep = WK_PAGEDEP(wk);
                        for (i = 0; i < DAHASHSZ; i++) {
                                if (LIST_FIRST(&pagedep->pd_diraddhd[i]) ==
                                    NULL)
                                        continue;
                                if ((error =
                                    flush_pagedep_deps(vp, pagedep->pd_mnt,
                                                &pagedep->pd_diraddhd[i]))) {
                                        FREE_LOCK(&lk);
                                        bawrite(bp);
                                        return (error);
                                }
                        }
                        break;

                case D_MKDIR:
                        /*
                         * This case should never happen if the vnode has
                         * been properly sync'ed. However, if this function
                         * is used at a place where the vnode has not yet
                         * been sync'ed, this dependency can show up. So,
                         * rather than panic, just flush it.
                         */
                        nbp = WK_MKDIR(wk)->md_buf;
                        KASSERT(bp != nbp);
                        gotit = getdirtybuf(nbp, waitfor);
                        if (gotit == 0)
                                break;
                        else if (gotit == -1)
                                goto loop;
                        FREE_LOCK(&lk);
                        if (waitfor == MNT_NOWAIT) {
                                bawrite(nbp);
                        } else if ((error = VOP_BWRITE(nbp)) != 0) {
                                bawrite(bp);
                                return (error);
                        }
                        ACQUIRE_LOCK(&lk);
                        break;

                case D_BMSAFEMAP:
                        /*
                         * This case should never happen if the vnode has
                         * been properly sync'ed. However, if this function
                         * is used at a place where the vnode has not yet
                         * been sync'ed, this dependency can show up. So,
                         * rather than panic, just flush it.
                         */
                        nbp = WK_BMSAFEMAP(wk)->sm_buf;
                        if (bp == nbp)
                                break;
                        gotit = getdirtybuf(nbp, waitfor);
                        if (gotit == 0)
                                break;
                        else if (gotit == -1)
                                goto loop;
                        FREE_LOCK(&lk);
                        if (waitfor == MNT_NOWAIT) {
                                bawrite(nbp);
                        } else if ((error = VOP_BWRITE(nbp)) != 0) {
                                bawrite(bp);
                                return (error);
                        }
                        ACQUIRE_LOCK(&lk);
                        break;

                default:
                        FREE_LOCK(&lk);
                        panic("softdep_sync_metadata: Unknown type %s",
                            TYPENAME(wk->wk_type));
                        /* NOTREACHED */
                }
        }
        do {
                nbp = LIST_NEXT(bp, b_vnbufs);
                gotit = getdirtybuf(nbp, MNT_WAIT);
        } while (gotit == -1);
        FREE_LOCK(&lk);
        bawrite(bp);
        ACQUIRE_LOCK(&lk);
        if (nbp != NULL) {
                bp = nbp;
                goto loop;
        }
        /*
         * The brief unlock is to allow any pent up dependency
         * processing to be done. Then proceed with the second pass.
         */
        if (waitfor == MNT_NOWAIT) {
                waitfor = MNT_WAIT;
                FREE_LOCK(&lk);
                ACQUIRE_LOCK(&lk);
                goto top;
        }

        /*
         * If we have managed to get rid of all the dirty buffers,
         * then we are done. For certain directories and block
         * devices, we may need to do further work.
         *
         * We must wait for any I/O in progress to finish so that
         * all potential buffers on the dirty list will be visible.
         */
        drain_output(vp, 1);
        if (LIST_EMPTY(&vp->v_dirtyblkhd)) {
                FREE_LOCK(&lk);
                return (0);
        }

        FREE_LOCK(&lk);
        /*
         * If we are trying to sync a block device, some of its buffers may
         * contain metadata that cannot be written until the contents of some
         * partially written files have been written to disk. The only easy
         * way to accomplish this is to sync the entire filesystem (luckily
         * this happens rarely).
         */
        if (vn_isdisk(vp, NULL) &&
            vp->v_specmountpoint && !VOP_ISLOCKED(vp) &&
            (error = VFS_SYNC(vp->v_specmountpoint, MNT_WAIT, 0, ap->a_cred,
             ap->a_p)) != 0)
                return (error);
        return (0);
}

/*
 * Flush the dependencies associated with an inodedep.
 * Called with splbio blocked.
 */
STATIC int
flush_inodedep_deps(struct fs *fs, ufsino_t ino)
{
	struct inodedep *inodedep;
	struct allocdirect *adp;
	int gotit, error, waitfor;
	struct buf *bp;

	splassert(IPL_BIO);

	/*
	 * This work is done in two passes. The first pass grabs most
	 * of the buffers and begins asynchronously writing them. The
	 * only way to wait for these asynchronous writes is to sleep
	 * on the filesystem vnode which may stay busy for a long time
	 * if the filesystem is active. So, instead, we make a second
	 * pass over the dependencies blocking on each write. In the
	 * usual case we will be blocking against a write that we
	 * initiated, so when it is done the dependency will have been
	 * resolved. Thus the second pass is expected to end quickly.
	 * We give a brief window at the top of the loop to allow
	 * any pending I/O to complete.
	 */
	for (waitfor = MNT_NOWAIT; ; ) {
	retry_ino:
		FREE_LOCK(&lk);
		ACQUIRE_LOCK(&lk);
		if (inodedep_lookup(fs, ino, 0, &inodedep) == 0)
			return (0);
		TAILQ_FOREACH(adp, &inodedep->id_inoupdt, ad_next) {
			if (adp->ad_state & DEPCOMPLETE)
				continue;
			bp = adp->ad_buf;
			gotit = getdirtybuf(bp, waitfor);
			if (gotit == 0) {
				if (waitfor == MNT_NOWAIT)
					continue;
				break;
			} else if (gotit == -1)
				goto retry_ino;
			FREE_LOCK(&lk);
			if (waitfor == MNT_NOWAIT) {
				bawrite(bp);
			} else if ((error = VOP_BWRITE(bp)) != 0) {
				ACQUIRE_LOCK(&lk);
				return (error);
			}
			ACQUIRE_LOCK(&lk);
			break;
		}
		if (adp != NULL)
			continue;
	retry_newino:
		TAILQ_FOREACH(adp, &inodedep->id_newinoupdt, ad_next) {
			if (adp->ad_state & DEPCOMPLETE)
				continue;
			bp = adp->ad_buf;
			gotit = getdirtybuf(bp, waitfor);
			if (gotit == 0) {
				if (waitfor == MNT_NOWAIT)
					continue;
				break;
			} else if (gotit == -1)
				goto retry_newino;
			FREE_LOCK(&lk);
			if (waitfor == MNT_NOWAIT) {
				bawrite(bp);
			} else if ((error = VOP_BWRITE(bp)) != 0) {
				ACQUIRE_LOCK(&lk);
				return (error);
			}
			ACQUIRE_LOCK(&lk);
			break;
		}
		if (adp != NULL)
			continue;
		/*
		 * If pass2, we are done, otherwise do pass 2.
		 */
		if (waitfor == MNT_WAIT)
			break;
		waitfor = MNT_WAIT;
	}
	/*
	 * Try freeing inodedep in case all dependencies have been removed.
	 */
	if (inodedep_lookup(fs, ino, 0, &inodedep) != 0)
		(void) free_inodedep(inodedep);
	return (0);
}

/*
 * Eliminate a pagedep dependency by flushing out all its diradd dependencies.
 * Called with splbio blocked.
 */
STATIC int
flush_pagedep_deps(struct vnode *pvp, struct mount *mp,
    struct diraddhd *diraddhdp)
{
	struct proc *p = CURPROC; /* XXX */
	struct worklist *wk;
	struct inodedep *inodedep;
	struct ufsmount *ump;
	struct diradd *dap;
	struct vnode *vp;
	int gotit, error = 0;
	struct buf *bp;
	ufsino_t inum;

	splassert(IPL_BIO);

	ump = VFSTOUFS(mp);
	while ((dap = LIST_FIRST(diraddhdp)) != NULL) {
		/*
		 * Flush ourselves if this directory entry
		 * has a MKDIR_PARENT dependency.
		 */
		if (dap->da_state & MKDIR_PARENT) {
			FREE_LOCK(&lk);
			if ((error = UFS_UPDATE(VTOI(pvp), 1)))
				break;
			ACQUIRE_LOCK(&lk);
			/*
			 * If that cleared dependencies, go on to next.
			 */
			if (dap != LIST_FIRST(diraddhdp))
				continue;
			if (dap->da_state & MKDIR_PARENT) {
				FREE_LOCK(&lk);
				panic("flush_pagedep_deps: MKDIR_PARENT");
			}
		}
		/*
		 * A newly allocated directory must have its "." and
		 * ".." entries written out before its name can be
		 * committed in its parent. We do not want or need
		 * the full semantics of a synchronous VOP_FSYNC as
		 * that may end up here again, once for each directory
		 * level in the filesystem. Instead, we push the blocks
		 * and wait for them to clear. We have to fsync twice
		 * because the first call may choose to defer blocks
		 * that still have dependencies, but deferral will
		 * happen at most once.
		 */
		inum = dap->da_newinum;
		if (dap->da_state & MKDIR_BODY) {
			FREE_LOCK(&lk);
			if ((error = VFS_VGET(mp, inum, &vp)) != 0)
				break;
			if ((error=VOP_FSYNC(vp, p->p_ucred, MNT_NOWAIT, p)) ||
			    (error=VOP_FSYNC(vp, p->p_ucred, MNT_NOWAIT, p))) {
				vput(vp);
				break;
			}
			drain_output(vp, 0);
			/*
			 * If first block is still dirty with a D_MKDIR
			 * dependency then it needs to be written now.
			 */
			for (;;) {
				error = 0;
				ACQUIRE_LOCK(&lk);
				bp = incore(vp, 0);
				if (bp == NULL) {
					FREE_LOCK(&lk);
					break;
				}
				LIST_FOREACH(wk, &bp->b_dep, wk_list)
					if (wk->wk_type == D_MKDIR)
						break;
				if (wk) {
					gotit = getdirtybuf(bp, MNT_WAIT);
					FREE_LOCK(&lk);
					if (gotit == -1)
						continue;
					if (gotit && (error = bwrite(bp)) != 0)
						break;
				} else
					FREE_LOCK(&lk);
				break;
			}
			vput(vp);
			/* Flushing of first block failed */
			if (error)
				break;
			ACQUIRE_LOCK(&lk);
			/*
			 * If that cleared dependencies, go on to next.
			 */
			if (dap != LIST_FIRST(diraddhdp))
				continue;
			if (dap->da_state & MKDIR_BODY) {
				FREE_LOCK(&lk);
				panic("flush_pagedep_deps: MKDIR_BODY");
			}
		}
		/*
		 * Flush the inode on which the directory entry depends.
		 * Having accounted for MKDIR_PARENT and MKDIR_BODY above,
		 * the only remaining dependency is that the updated inode
		 * count must get pushed to disk. The inode has already
		 * been pushed into its inode buffer (via VOP_UPDATE) at
		 * the time of the reference count change. So we need only
		 * locate that buffer, ensure that there will be no rollback
		 * caused by a bitmap dependency, then write the inode buffer.
		 */
		if (inodedep_lookup(ump->um_fs, inum, 0, &inodedep) == 0) {
			FREE_LOCK(&lk);
			panic("flush_pagedep_deps: lost inode");
		}
		/*
		 * If the inode still has bitmap dependencies,
		 * push them to disk.
		 */
	retry:
		if ((inodedep->id_state & DEPCOMPLETE) == 0) {
			bp = inodedep->id_buf;
			gotit = getdirtybuf(bp, MNT_WAIT);
			if (gotit == -1)
				goto retry;
			FREE_LOCK(&lk);
			if (gotit && (error = bwrite(bp)) != 0)
				break;
			ACQUIRE_LOCK(&lk);
			if (dap != LIST_FIRST(diraddhdp))
				continue;
		}
		/*
		 * If the inode is still sitting in a buffer waiting
		 * to be written, push it to disk.
		 */
		FREE_LOCK(&lk);
		if ((error = bread(ump->um_devvp,
		    fsbtodb(ump->um_fs, ino_to_fsba(ump->um_fs, inum)),
		    (int)ump->um_fs->fs_bsize, &bp)) != 0) {
			brelse(bp);
			break;
		}
		if ((error = bwrite(bp)) != 0)
			break;
		ACQUIRE_LOCK(&lk);
		/*
		 * If we have failed to get rid of all the dependencies
		 * then something is seriously wrong.
		 */
		if (dap == LIST_FIRST(diraddhdp)) {
			FREE_LOCK(&lk);
			panic("flush_pagedep_deps: flush failed");
		}
	}
	if (error)
		ACQUIRE_LOCK(&lk);
	return (error);
}

/*
 * A large burst of file addition or deletion activity can drive the
 * memory load excessively high. First attempt to slow things down
 * using the techniques below. If that fails, this routine requests
 * the offending operations to fall back to running synchronously
 * until the memory load returns to a reasonable level.
 */
int
softdep_slowdown(struct vnode *vp)
{
	int max_softdeps_hard;

	max_softdeps_hard = max_softdeps * 11 / 10;
	if (num_dirrem < max_softdeps_hard / 2 &&
	    num_inodedep < max_softdeps_hard)
		return (0);
	stat_sync_limit_hit += 1;
	return (1);
}

/*
 * If memory utilization has gotten too high, deliberately slow things
 * down and speed up the I/O processing.
 */
STATIC int
request_cleanup(int resource, int islocked)
{
	struct proc *p = CURPROC;
	int s;

	/*
	 * We never hold up the filesystem syncer process.
	 */
	if (p == filesys_syncer || (p->p_flag & P_SOFTDEP))
		return (0);
	/*
	 * First check to see if the work list has gotten backlogged.
	 * If it has, co-opt this process to help clean up two entries.
	 * Because this process may hold inodes locked, we cannot
	 * handle any remove requests that might block on a locked
	 * inode as that could lead to deadlock. We set P_SOFTDEP
	 * to avoid recursively processing the worklist.
	 */
	if (num_on_worklist > max_softdeps / 10) {
		atomic_setbits_int(&p->p_flag, P_SOFTDEP);
		if (islocked)
			FREE_LOCK(&lk);
		process_worklist_item(NULL, NULL, LK_NOWAIT);
```

```c
                process_worklist_item(NULL, NULL, LK_NOWAIT);
                atomic_clearbits_int(&p->p_flag, P_SOFTDEP);
                stat_worklist_push += 2;
                if (islocked)
                        ACQUIRE_LOCK(&lk);
                return(1);
        }
        /*
         * Next, we attempt to speed up the syncer process. If that
         * is successful, then we allow the process to continue.
         */
        if (speedup_syncer())
                return(0);
        /*
         * If we are resource constrained on inode dependencies, try
         * flushing some dirty inodes. Otherwise, we are constrained
         * by file deletions, so try accelerating flushes of directories
         * with removal dependencies. We would like to do the cleanup
         * here, but we probably hold an inode locked at this point and
         * that might deadlock against one that we try to clean. So,
         * the best that we can do is request the syncer daemon to do
         * the cleanup for us.
         */
        switch (resource) {

        case FLUSH_INODES:
                stat_ino_limit_push += 1;
                req_clear_inodedeps += 1;
                stat_countp = &stat_ino_limit_hit;
                break;

        case FLUSH_REMOVE:
                stat_blk_limit_push += 1;
                req_clear_remove += 1;
                stat_countp = &stat_blk_limit_hit;
                break;

        default:
                if (islocked)
                        FREE_LOCK(&lk);
                panic("request_cleanup: unknown type");
        }
        /*
         * Hopefully the syncer daemon will catch up and awaken us.
         * We wait at most tickdelay before proceeding in any case.
         */
        if (islocked == 0)
                ACQUIRE_LOCK(&lk);
        proc_waiting += 1;
        if (!timeout_pending(&proc_waiting_timeout))
                timeout_add(&proc_waiting_timeout, tickdelay > 2 ? tickdelay : 2);

        s = FREE_LOCK_INTERLOCKED(&lk);
        tsleep_nsec(&proc_waiting, PPAUSE, "softupdate", INFSLP);
        ACQUIRE_LOCK_INTERLOCKED(&lk, s);
        proc_waiting -= 1;
        if (islocked == 0)
                FREE_LOCK(&lk);
        return (1);
}

/*
 * Awaken processes pausing in request_cleanup and clear proc_waiting
 * to indicate that there is no longer a timer running.
 */
void
pause_timer(void *arg)
{

        *stat_countp += 1;
        wakeup_one(&proc_waiting);
        if (proc_waiting > 0)
                timeout_add(&proc_waiting_timeout, tickdelay > 2 ? tickdelay : 2);
}

/*
 * Flush out a directory with at least one removal dependency in an effort to
 * reduce the number of dirrem, freefile, and freeblks dependency structures.
 */
STATIC void
clear_remove(struct proc *p)
{
        struct pagedep_hashhead *pagedephd;
        struct pagedep *pagedep;
        static int next = 0;
        struct mount *mp;
        struct vnode *vp;
        int error, cnt;
        ufsino_t ino;

        ACQUIRE_LOCK(&lk);
        for (cnt = 0; cnt <= pagedep_hash; cnt++) {
                pagedephd = &pagedep_hashtbl[next++];
                if (next > pagedep_hash)
                        next = 0;
                LIST_FOREACH(pagedep, pagedephd, pd_hash) {
                        if (LIST_FIRST(&pagedep->pd_dirremhd) == NULL)
                                continue;
                        mp = pagedep->pd_mnt;
                        ino = pagedep->pd_ino;
#if 0
                        if (vn_start_write(NULL, &mp, V_NOWAIT) != 0)
                                continue;
#endif
                        FREE_LOCK(&lk);
                        if ((error = VFS_VGET(mp, ino, &vp)) != 0) {
                                softdep_error("clear_remove: vget", error);
#if 0
                                vn_finished_write(mp);
#endif
                                return;
                        }
                        if ((error = VOP_FSYNC(vp, p->p_ucred, MNT_NOWAIT, p)))
                                softdep_error("clear_remove: fsync", error);
                        drain_output(vp, 0);
                        vput(vp);
#if 0
                        vn_finished_write(mp);
#endif
                        return;
                }
        }
        FREE_LOCK(&lk);
}

/*
 * Clear out a block of dirty inodes in an effort to reduce
 * the number of inodedep dependency structures.
 */
STATIC void
clear_inodedeps(struct proc *p)
{
        struct inodedep_hashhead *inodedephd;
        struct inodedep *inodedep = NULL;
        static int next = 0;
        struct mount *mp;
        struct vnode *vp;
        struct fs *fs;
        int error, cnt;
        ufsino_t firstino, lastino, ino;

        ACQUIRE_LOCK(&lk);
        /*
         * Pick a random inode dependency to be cleared.
         * We will then gather up all the inodes in its block
         * that have dependencies and flush them out.
         */
        for (cnt = 0; cnt <= inodedep_hash; cnt++) {
                inodedephd = &inodedep_hashtbl[next++];
                if (next > inodedep_hash)
                        next = 0;
                if ((inodedep = LIST_FIRST(inodedephd)) != NULL)
                        break;
        }
        if (inodedep == NULL) {
                FREE_LOCK(&lk);
                return;
        }
        /*
         * Ugly code to find mount point given pointer to superblock.
         */
        fs = inodedep->id_fs;
        TAILQ_FOREACH(mp, &mountlist, mnt_list)
                if ((mp->mnt_flag & MNT_SOFTDEP) && fs == VFSTOUFS(mp)->um_fs)
                        break;
        /*
         * Find the last inode in the block with dependencies.
         */
        firstino = inodedep->id_ino & ~(INOPB(fs) - 1);
        for (lastino = firstino + INOPB(fs) - 1; lastino > firstino; lastino--)
                if (inodedep_lookup(fs, lastino, 0, &inodedep) != 0)
                        break;
        /*
         * Asynchronously push all but the last inode with dependencies.
         * Synchronously push the last inode with dependencies to ensure
         * that the inode block gets written to free up the inodedeps.
         */
        for (ino = firstino; ino <= lastino; ino++) {
                if (inodedep_lookup(fs, ino, 0, &inodedep) == 0)
                        continue;
                FREE_LOCK(&lk);
#if 0
                if (vn_start_write(NULL, &mp, V_NOWAIT) != 0)
                        continue;
#endif
                if ((error = VFS_VGET(mp, ino, &vp)) != 0) {
                        softdep_error("clear_inodedeps: vget", error);
#if 0
                        vn_finished_write(mp);
#endif
                        return;
                }
                if (ino == lastino) {
                        if ((error = VOP_FSYNC(vp, p->p_ucred, MNT_WAIT, p)))
                                softdep_error("clear_inodedeps: fsync1", error);
                } else {
                        if ((error = VOP_FSYNC(vp, p->p_ucred, MNT_NOWAIT, p)))
                                softdep_error("clear_inodedeps: fsync2", error);
                        drain_output(vp, 0);
                }
                vput(vp);
#if 0
                vn_finished_write(mp);
#endif
                ACQUIRE_LOCK(&lk);
        }
        FREE_LOCK(&lk);
}

/*
 * Function to determine if the buffer has outstanding dependencies
 * that will cause a roll-back if the buffer is written. If wantcount
 * is set, return number of dependencies, otherwise just yes or no.
 */
int
softdep_count_dependencies(struct buf *bp, int wantcount, int islocked)
{
        struct worklist *wk;
        struct inodedep *inodedep;
        struct indirdep *indirdep;
        struct allocindir *aip;
        struct pagedep *pagedep;
        struct diradd *dap;
        int i, retval;

        retval = 0;
        if (!islocked)
                ACQUIRE_LOCK(&lk);
        LIST_FOREACH(wk, &bp->b_dep, wk_list) {
                switch (wk->wk_type) {

                case D_INODEDEP:
                        inodedep = WK_INODEDEP(wk);
                        if ((inodedep->id_state & DEPCOMPLETE) == 0) {
                                /* bitmap allocation dependency */
                                retval += 1;
                                if (!wantcount)
                                        goto out;
                        }
                        if (TAILQ_FIRST(&inodedep->id_inoupdt)) {
                                /* direct block pointer dependency */
                                retval += 1;
                                if (!wantcount)
                                        goto out;
                        }
                        continue;

                case D_INDIRDEP:
                        indirdep = WK_INDIRDEP(wk);

                        LIST_FOREACH(aip, &indirdep->ir_deplisthd, ai_next) {
                                /* indirect block pointer dependency */
                                retval += 1;
                                if (!wantcount)
                                        goto out;
                        }
                        continue;

                case D_PAGEDEP:
                        pagedep = WK_PAGEDEP(wk);
                        for (i = 0; i < DAHASHSZ; i++) {

                                LIST_FOREACH(dap, &pagedep->pd_diraddhd[i], da_pdlist) {
                                        /* directory entry dependency */
                                        retval += 1;
                                        if (!wantcount)
                                                goto out;
                                }
                        }
                        continue;

                case D_BMSAFEMAP:
                case D_ALLOCDIRECT:
                case D_ALLOCINDIR:
                case D_MKDIR:
                        /* never a dependency on these blocks */
                        continue;

                default:
                        if (!islocked)
                                FREE_LOCK(&lk);
                        panic("softdep_check_for_rollback: Unexpected type %s",
                            TYPENAME(wk->wk_type));
                        /* NOTREACHED */
                }
        }
out:
        if (!islocked)
                FREE_LOCK(&lk);
        return retval;
}

/*
 * Acquire exclusive access to a buffer.
 * Must be called with splbio blocked.
 * Returns:
 * 1 if the buffer was acquired and is dirty;
 * 0 if the buffer was clean, or we would have slept but had MN_NOWAIT;
 * -1 if we slept and may try again (but not with this bp).
 */
STATIC int
getdirtybuf(struct buf *bp, int waitfor)
{
        int s;
	if (bp == NULL)
		return (0);

	splassert(IPL_BIO);

	if (bp->b_flags & B_BUSY) {
		if (waitfor != MNT_WAIT)
			return (0);
		bp->b_flags |= B_WANTED;
		s = FREE_LOCK_INTERLOCKED(&lk);
		tsleep_nsec(bp, PRIBIO+1, "sdsdty", INFSLP);
		ACQUIRE_LOCK_INTERLOCKED(&lk, s);
		return (-1);
	}
	if ((bp->b_flags & B_DELWRI) == 0)
		return (0);
	bremfree(bp);
	buf_acquire(bp);
	return (1);
}

/*
 * Wait for pending output on a vnode to complete.
 * Must be called with vnode locked.
 */
STATIC void
drain_output(struct vnode *vp, int islocked)
{
	int s;

	if (!islocked)
		ACQUIRE_LOCK(&lk);

	splassert(IPL_BIO);

	while (vp->v_numoutput) {
		vp->v_bioflag |= VBIOWAIT;
		s = FREE_LOCK_INTERLOCKED(&lk);
		tsleep_nsec(&vp->v_numoutput, PRIBIO+1, "drain_output", INFSLP);
		ACQUIRE_LOCK_INTERLOCKED(&lk, s);
	}
	if (!islocked)
		FREE_LOCK(&lk);
}

/*
 * Called whenever a buffer that is being invalidated or reallocated
 * contains dependencies. This should only happen if an I/O error has
 * occurred. The routine is called with the buffer locked.
 */
void
softdep_deallocate_dependencies(struct buf *bp)
{

	if ((bp->b_flags & B_ERROR) == 0)
		panic("softdep_deallocate_dependencies: dangling deps");
	softdep_error(bp->b_vp->v_mount->mnt_stat.f_mntonname, bp->b_error);
	panic("softdep_deallocate_dependencies: unrecovered I/O error");
}

/*
 * Function to handle asynchronous write errors in the filesystem.
 */
void
softdep_error(char *func, int error)
{

	/* XXX should do something better! */
	printf("%s: got error %d while accessing filesystem\n", func, error);
}

#ifdef DDB
#include <machine/db_machdep.h>
#include <ddb/db_interface.h>
#include <ddb/db_output.h>

void
softdep_print(struct buf *bp, int full,
    int (*pr)(const char *, ...) __attribute__((__format__(__kprintf__,1,2))))
{
	struct worklist *wk;

	(*pr)("  deps:\n");
	LIST_FOREACH(wk, &bp->b_dep, wk_list)
		worklist_print(wk, full, pr);
}

void
worklist_print(struct worklist *wk, int full,
    int (*pr)(const char *, ...) __attribute__((__format__(__kprintf__,1,2))))
{
	struct pagedep *pagedep;
	struct inodedep *inodedep;
	struct newblk *newblk;
	struct bmsafemap *bmsafemap;
	struct allocdirect *adp;
	struct indirdep *indirdep;
	struct allocindir *aip;
	struct freefrag *freefrag;
	struct freeblks *freeblks;
	struct freefile *freefile;
	struct diradd *dap;
	struct mkdir *mkdir;
	struct dirrem *dirrem;
	struct newdirblk *newdirblk;
	char prefix[33];
	int i;

	for (prefix[i = 2 * MIN(16, full)] = '\0'; i--; prefix[i] = ' ')
		;

	(*pr)("%s%s(%p) state %b\n%s", prefix, TYPENAME(wk->wk_type), wk,
	    wk->wk_state, DEP_BITS, prefix);
	switch (wk->wk_type) {
	case D_PAGEDEP:
		pagedep = WK_PAGEDEP(wk);
		(*pr)("mount %p ino %u lbn %lld\n", pagedep->pd_mnt,
		    pagedep->pd_ino, (long long)pagedep->pd_lbn);
		break;
	case D_INODEDEP:
		inodedep = WK_INODEDEP(wk);
		(*pr)("fs %p ino %u nlinkdelta %u dino %p\n"
		    "%s  bp %p savsz %lld\n", inodedep->id_fs,
		    inodedep->id_ino, inodedep->id_nlinkdelta,
		    inodedep->id_un.idu_savedino1,
		    prefix, inodedep->id_buf, inodedep->id_savedsize);
		break;
	case D_NEWBLK:
		newblk = WK_NEWBLK(wk);
		(*pr)("fs %p newblk %lld state %d bmsafemap %p\n",
		    newblk->nb_fs, (long long)newblk->nb_newblkno,
		    newblk->nb_state, newblk->nb_bmsafemap);
		break;
	case D_BMSAFEMAP:
		bmsafemap = WK_BMSAFEMAP(wk);
		(*pr)("buf %p\n", bmsafemap->sm_buf);
		break;
	case D_ALLOCDIRECT:
		adp = WK_ALLOCDIRECT(wk);
		(*pr)("lbn %lld newlbk %lld oldblk %lld newsize %ld olsize "
		    "%ld\n%s  bp %p inodedep %p freefrag %p\n",
		    (long long)adp->ad_lbn, (long long)adp->ad_newblkno,
		    (long long)adp->ad_oldblkno, adp->ad_newsize,
		    adp->ad_oldsize,
		    prefix, adp->ad_buf, adp->ad_inodedep, adp->ad_freefrag);
		break;
	case D_INDIRDEP:
		indirdep = WK_INDIRDEP(wk);
		(*pr)("savedata %p savebp %p\n", indirdep->ir_saveddata,
		    indirdep->ir_savebp);
		break;
	case D_ALLOCINDIR:
		aip = WK_ALLOCINDIR(wk);
		(*pr)("off %d newblk %lld oldblk %lld freefrag %p\n"
		    "%s  indirdep %p buf %p\n", aip->ai_offset,
		    (long long)aip->ai_newblkno, (long long)aip->ai_oldblkno,
		    aip->ai_freefrag, prefix, aip->ai_indirdep, aip->ai_buf);
		break;
	case D_FREEFRAG:
		freefrag = WK_FREEFRAG(wk);
		(*pr)("vnode %p mp %p blkno %lld fsize %ld ino %u\n",
		    freefrag->ff_devvp, freefrag->ff_mnt,
		    (long long)freefrag->ff_blkno, freefrag->ff_fragsize,
		    freefrag->ff_inum);
		break;
	case D_FREEBLKS:
		freeblks = WK_FREEBLKS(wk);
		(*pr)("previno %u devvp %p mp %p oldsz %lld newsz %lld\n"
		    "%s  chkcnt %d uid %d\n", freeblks->fb_previousinum,
		    freeblks->fb_devvp, freeblks->fb_mnt, freeblks->fb_oldsize,
		    freeblks->fb_newsize,
		    prefix, freeblks->fb_chkcnt, freeblks->fb_uid);
		break;
	case D_FREEFILE:
		freefile = WK_FREEFILE(wk);
		(*pr)("mode %x oldino %u vnode %p mp %p\n", freefile->fx_mode,
		    freefile->fx_oldinum, freefile->fx_devvp, freefile->fx_mnt);
		break;
	case D_DIRADD:
		dap = WK_DIRADD(wk);
		(*pr)("off %d ino %u da_un %p\n", dap->da_offset,
		    dap->da_newinum, dap->da_un.dau_previous);
		break;
	case D_MKDIR:
		mkdir = WK_MKDIR(wk);
		(*pr)("diradd %p bp %p\n", mkdir->md_diradd, mkdir->md_buf);
		break;
	case D_DIRREM:
		dirrem = WK_DIRREM(wk);
		(*pr)("mp %p ino %u dm_un %p\n", dirrem->dm_mnt,
		    dirrem->dm_oldinum, dirrem->dm_un.dmu_pagedep);
		break;
	case D_NEWDIRBLK:
		newdirblk = WK_NEWDIRBLK(wk);
		(*pr)("pagedep %p\n", newdirblk->db_pagedep);
		break;
	}
}
#endif
File: ffs_softdep_stub.c


/*	$OpenBSD: ffs_softdep_stub.c,v 1.18 2013/06/11 16:42:18 deraadt Exp $	*/

/*
 * Copyright 1998 Marshall Kirk McKusick. All Rights Reserved.
 *
 * The soft updates code is derived from the appendix of a University
 * of Michigan technical report (Gregory R. Ganger and Yale N. Patt,
 * "Soft Updates: A Solution to the Metadata Update Problem in File
 * Systems", CSE-TR-254-95, August 1995).
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions
 * are met:
 * 1. Redistributions of source code must retain the above copyright
 *    notice, this list of conditions and the following disclaimer.
 * 2. Redistributions in binary form must reproduce the above copyright
 *    notice, this list of conditions and the following disclaimer in the
 *    documentation and/or other materials provided with the distribution.
 * 3. None of the names of McKusick, Ganger, or the University of Michigan
 *    may be used to endorse or promote products derived from this software
 *    without specific prior written permission.
 *
 * THIS SOFTWARE IS PROVIDED BY MARSHALL KIRK MCKUSICK ``AS IS'' AND
 * ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
 * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
 * ARE DISCLAIMED.  IN NO EVENT SHALL MARSHALL KIRK MCKUSICK BE LIABLE
 * FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
 * DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
 * OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
 * HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
 * LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
 * OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
 * SUCH DAMAGE.
 *
 *	from: @(#)ffs_softdep_stub.c	9.1 (McKusick) 7/10/97
 * $FreeBSD: src/sys/ufs/ffs/ffs_softdep_stub.c,v 1.14 2000/08/09 00:41:54 tegge Exp $
 */

#ifndef FFS_SOFTUPDATES

#include <sys/param.h>
#include <sys/vnode.h>
#include <sys/systm.h>
#include <ufs/ufs/quota.h>
#include <ufs/ufs/inode.h>
#include <ufs/ffs/ffs_extern.h>
#include <ufs/ufs/ufs_extern.h>

int
softdep_flushfiles(struct mount *oldmnt, int flags, struct proc *p)
{
	panic("softdep_flushfiles called");
}

int
softdep_mount(struct vnode *devvp, struct mount *mp, struct fs *fs,
    struct ucred *cred)
{
	return (0);
}

void
softdep_initialize(void)
{
	return;
}

#ifndef __OPTIMIZE__

void
softdep_setup_inomapdep(struct buf *bp, struct inode *ip, ufsino_t newinum)
{
	panic("softdep_setup_inomapdep called");
}

void
softdep_setup_blkmapdep(struct buf *bp, struct fs *fs, daddr_t newblkno)
{
	panic("softdep_setup_blkmapdep called");
}

void
softdep_setup_allocdirect(struct inode *ip, daddr_t lbn, daddr_t newblkno,
    daddr_t oldblkno, long newsize, long oldsize, struct buf *bp)
{
	panic("softdep_setup_allocdirect called");
}

void
softdep_setup_allocindir_page(struct inode *ip, daddr_t lbn, struct buf *bp,
    int ptrno, daddr_t newblkno, daddr_t oldblkno, struct buf *nbp)
{
	panic("softdep_setup_allocindir_page called");
}

void
softdep_setup_allocindir_meta(struct buf *nbp, struct inode *ip,
    struct buf *bp, int ptrno, daddr_t newblkno)
{
	panic("softdep_setup_allocindir_meta called");
}
```

```c
void
softdep_setup_freeblocks(struct inode *ip, off_t length)
{
        panic("softdep_setup_freeblocks called");
}

void
softdep_freefile(struct vnode *pvp, ufsino_t ino, mode_t mode)
{
        panic("softdep_freefile called");
}

int
softdep_setup_directory_add(struct buf *bp, struct inode *dp, off_t diroffset,
    long newinum, struct buf *newdirbp, int isnewblk)
{
        panic("softdep_setup_directory_add called");
        return (0);
}

void
softdep_change_directoryentry_offset(struct inode *dp, caddr_t base,
    caddr_t oldloc, caddr_t newloc, int entrysize)
{
        panic("softdep_change_directoryentry_offset called");
}

void
softdep_setup_remove(struct buf *bp, struct inode *dp, struct inode *ip,
    int isrmdir)
{
        panic("softdep_setup_remove called");
}

void
softdep_setup_directory_change(struct buf *bp, struct inode *dp,
    struct inode *ip, long newinum, int isrmdir)
{
        panic("softdep_setup_directory_change called");
}

void
softdep_change_linkcnt(struct inode *ip, int nodelay)
{
        panic("softdep_change_linkcnt called");
}

void
softdep_load_inodeblock(struct inode *ip)
{
        panic("softdep_load_inodeblock called");
}

void
softdep_update_inodeblock(struct inode *ip, struct buf *bp, int waitfor)
{
        panic("softdep_update_inodeblock called");
}

#endif

void
softdep_fsync_mountdev(struct vnode *vp, int waitfor)
{
        return;
}

int
softdep_flushworklist(struct mount *oldmnt, int *countp, struct proc *p)
{
        *countp = 0;
        return (0);
}

int
softdep_sync_metadata(struct vop_fsync_args *ap)
{
        return (0);
}

#ifndef __OPTIMIZE__

int
softdep_slowdown(struct vnode *vp)
{
        panic("softdep_slowdown called");
}

#endif

#endif /* !FFS_SOFTUPDATES */
File: ffs_subr.c


/*      $OpenBSD: ffs_subr.c,v 1.32 2016/08/10 11:33:01 natano Exp $        */
/*      $NetBSD: ffs_subr.c,v 1.6 1996/03/17 02:16:23 christos Exp $        */

/*
 * Copyright (c) 1982, 1986, 1989, 1993
 *      The Regents of the University of California.  All rights reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions
 * are met:
 * 1. Redistributions of source code must retain the above copyright
 *    notice, this list of conditions and the following disclaimer.
 * 2. Redistributions in binary form must reproduce the above copyright
 *    notice, this list of conditions and the following disclaimer in the
 *    documentation and/or other materials provided with the distribution.
 * 3. Neither the name of the University nor the names of its contributors
 *    may be used to endorse or promote products derived from this software
 *    without specific prior written permission.
 *
 * THIS SOFTWARE IS PROVIDED BY THE REGENTS AND CONTRIBUTORS ``AS IS'' AND
 * ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
 * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
 * ARE DISCLAIMED.  IN NO EVENT SHALL THE REGENTS OR CONTRIBUTORS BE LIABLE
 * FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
 * DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
 * OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
 * HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
 * LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
 * OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
 * SUCH DAMAGE.
 *
 *      @(#)ffs_subr.c   8.2 (Berkeley) 9/21/93
 */

#include <sys/param.h>
#include <ufs/ffs/fs.h>

#ifdef _KERNEL
#include <sys/systm.h>
#include <sys/vnode.h>
#include <sys/mount.h>
#include <sys/buf.h>

#include <ufs/ufs/quota.h>
#include <ufs/ufs/inode.h>
#include <ufs/ufs/ufsmount.h>
#include <ufs/ufs/ufs_extern.h>

#include <ufs/ffs/ffs_extern.h>

/*
 * Return buffer with the contents of block "offset" from the beginning of
 * directory "ip".  If "res" is non-zero, fill it in with a pointer to the
 * remaining space in the directory.
 */
int
ffs_bufatoff(struct inode *ip, off_t offset, char **res, struct buf **bpp)
{
        struct fs *fs;
        struct vnode *vp;
        struct buf *bp;
        daddr_t lbn;
        int bsize, error;

        vp = ITOV(ip);
        fs = ip->i_fs;
        lbn = lblkno(fs, offset);
        bsize = blksize(fs, ip, lbn);

        *bpp = NULL;
        if ((error = bread(vp, lbn, fs->fs_bsize, &bp)) != 0) {
                brelse(bp);
                return (error);
        }
        buf_adjcnt(bp, bsize);
        if (res)
                *res = (char *)bp->b_data + blkoff(fs, offset);
        *bpp = bp;
        return (0);
}
#else
/* Prototypes for userland */
void    ffs_fragacct(struct fs *, int, int32_t[], int);
int     ffs_isfreeblock(struct fs *, u_char *, daddr_t);
int     ffs_isblock(struct fs *, u_char *, daddr_t);
void    ffs_clrblock(struct fs *, u_char *, daddr_t);
void    ffs_setblock(struct fs *, u_char *, daddr_t);
__dead void panic(const char *, ...);
#endif

/*
 * Update the frsum fields to reflect addition or deletion
 * of some frags.
 */
void
ffs_fragacct(struct fs *fs, int fragmap, int32_t fraglist[], int cnt)
{
        int inblk;
        int field, subfield;
        int siz, pos;

        inblk = (int)(fragtbl[fs->fs_frag][fragmap]) << 1;
        fragmap <<= 1;
        for (siz = 1; siz < fs->fs_frag; siz++) {
                if ((inblk & (1 << (siz + (fs->fs_frag % NBBY)))) == 0)
                        continue;
                field = around[siz];
                subfield = inside[siz];
                for (pos = siz; pos <= fs->fs_frag; pos++) {
                        if ((fragmap & field) == subfield) {
                                fraglist[siz] += cnt;
                                pos += siz;
                                field <<= siz;
                                subfield <<= siz;
                        }
                        field <<= 1;
                        subfield <<= 1;
                }
        }
}

#if defined(_KERNEL) && defined(DIAGNOSTIC)
void
ffs_checkoverlap(struct buf *bp, struct inode *ip)
{
        daddr_t start, last;
        struct vnode *vp;
        struct buf *ep;

        start = bp->b_blkno;
        last = start + btodb(bp->b_bcount) - 1;
        LIST_FOREACH(ep, &bufhead, b_list) {
                if (ep == bp || (ep->b_flags & B_INVAL) ||
                    ep->b_vp == NULLVP)
                        continue;
                if (VOP_BMAP(ep->b_vp, 0, &vp, NULL, NULL))
                        continue;
                if (vp != ip->i_devvp)
                        continue;
                /* look for overlap */
                if (ep->b_bcount == 0 || ep->b_blkno > last ||
                    ep->b_blkno + btodb(ep->b_bcount) <= start)
                        continue;
                vprint("Disk overlap", vp);
                (void)printf("\tstart %lld, end %lld overlap start %llu, "
                    "end %llu\n", (long long)start, (long long)last,
                    (long long)ep->b_blkno,
                    (long long)(ep->b_blkno + btodb(ep->b_bcount) - 1));
                panic("Disk buffer overlap");
        }
}
#endif /* DIAGNOSTIC */

/*
 * block operations
 *
 * check if a block is available
 */
int
ffs_isblock(struct fs *fs, u_char *cp, daddr_t h)
{
        u_char mask;

        switch (fs->fs_frag) {
        default:
        case 8:
                return (cp[h] == 0xff);
        case 4:
                mask = 0x0f << ((h & 0x1) << 2);
                return ((cp[h >> 1] & mask) == mask);
        case 2:
                mask = 0x03 << ((h & 0x3) << 1);
                return ((cp[h >> 2] & mask) == mask);
        case 1:
                mask = 0x01 << (h & 0x7);
                return ((cp[h >> 3] & mask) == mask);
        }
}

/*
 * take a block out of the map
 */
void
ffs_clrblock(struct fs *fs, u_char *cp, daddr_t h)
{

        switch (fs->fs_frag) {
        default:
        case 8:
                cp[h] = 0;
                return;
        case 4:
                cp[h >> 1] &= ~(0x0f << ((h & 0x1) << 2));
                return;
        case 2:
                cp[h >> 2] &= ~(0x03 << ((h & 0x3) << 1));
                return;
        case 1:
                cp[h >> 3] &= ~(0x01 << (h & 0x7));
                return;
        }
}

/*
 * put a block into the map
 */
void
ffs_setblock(struct fs *fs, u_char *cp, daddr_t h)
{
        switch (fs->fs_frag) {
        default:
        case 8:
                cp[h] = 0xff;
                return;
        case 4:
                cp[h >> 1] |= (0x0f << ((h & 0x1) << 2));
                return;
        case 2:
                cp[h >> 2] |= (0x03 << ((h & 0x3) << 1));
                return;
        case 1:
                cp[h >> 3] |= (0x01 << (h & 0x7));
                return;
        }
}

/*
 * check if a block is free
 */
int
ffs_isfreeblock(struct fs *fs, u_char *cp, daddr_t h)
{

        switch (fs->fs_frag) {
        default:
        case 8:
                return (cp[h] == 0);
        case 4:
                return ((cp[h >> 1] & (0x0f << ((h & 0x1) << 2))) == 0);
        case 2:
                return ((cp[h >> 2] & (0x03 << ((h & 0x3) << 1))) == 0);
        case 1:
                return ((cp[h >> 3] & (0x01 << (h & 0x7))) == 0);
        }
}

#ifdef _KERNEL
/*
 * Initialize the vnode associated with a new inode, handle aliased
 * vnodes.
 */
int
ffs_vinit(struct mount *mntp, struct vnode **vpp)
{
        struct inode *ip;
        struct vnode *vp, *nvp;
        struct timeval mtv;

        vp = *vpp;
        ip = VTOI(vp);
        switch(vp->v_type = IFTOVT(DIP(ip, mode))) {
        case VCHR:
        case VBLK:
                vp->v_op = &ffs_specvops;
                if ((nvp = checkalias(vp, DIP(ip, rdev), mntp)) != NULL) {
                        /*
                         * Discard unneeded vnode, but save its inode.
                         * Note that the lock is carried over in the inode
                         * to the replacement vnode.
                         */
                        nvp->v_data = vp->v_data;
                        vp->v_data = NULL;
                        vp->v_op = &spec_vops;
#ifdef VFSLCKDEBUG
                        vp->v_flag &= ~VLOCKSWORK;
#endif
                        vrele(vp);
                        vgone(vp);
                        /*
                         * Reinitialize aliased inode.
                         */
                        vp = nvp;
                        ip->i_vnode = vp;
                }
                break;
        case VFIFO:
#ifdef FIFO
                vp->v_op = &ffs_fifovops;
                break;
#else
                return (EOPNOTSUPP);
#endif
        case VNON:
        case VBAD:
        case VSOCK:
        case VLNK:
        case VDIR:
        case VREG:
                break;
        }
        if (ip->i_number == ROOTINO)
                vp->v_flag |= VROOT;
        /*
         * Initialize modrev times
         */
        getmicrouptime(&mtv);
        ip->i_modrev = (u_quad_t)mtv.tv_sec << 32;
        ip->i_modrev |= (u_quad_t)mtv.tv_usec * 4294;
        *vpp = vp;
        return (0);
}
#endif /* _KERNEL */
File: ffs_tables.c


/*      $OpenBSD: ffs_tables.c,v 1.6 2011/07/03 18:23:10 tedu Exp $*/
/*      $NetBSD: ffs_tables.c,v 1.2 1994/06/29 06:46:35 cgd Exp $  */

/*
 * Copyright (c) 1982, 1986, 1993
 *      The Regents of the University of California.  All rights reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions
 * are met:
 * 1. Redistributions of source code must retain the above copyright
 *    notice, this list of conditions and the following disclaimer.
 * 2. Redistributions in binary form must reproduce the above copyright
 *    notice, this list of conditions and the following disclaimer in the
 *    documentation and/or other materials provided with the distribution.
 * 3. Neither the name of the University nor the names of its contributors
 *    may be used to endorse or promote products derived from this software
 *    without specific prior written permission.
 *
 * THIS SOFTWARE IS PROVIDED BY THE REGENTS AND CONTRIBUTORS ``AS IS'' AND
 * ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
 * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
 * ARE DISCLAIMED.  IN NO EVENT SHALL THE REGENTS OR CONTRIBUTORS BE LIABLE
 * FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
 * DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
 * OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
 * HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
 * LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
 * OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
 * SUCH DAMAGE.
 *
 *      @(#)ffs_tables.c 8.1 (Berkeley) 6/11/93
 */

#include <sys/param.h>
#include <ufs/ffs/fs.h>

/*
 * Bit patterns for identifying fragments in the block map
 * used as ((map & around) == inside)
 */
const int around[9] = {
        0x3, 0x7, 0xf, 0x1f, 0x3f, 0x7f, 0xff, 0x1ff, 0x3ff
};
const int inside[9] = {
        0x0, 0x2, 0x6, 0xe, 0x1e, 0x3e, 0x7e, 0xfe, 0x1fe
};

/*
 * Given a block map bit pattern, the frag tables tell whether a
 * particular size fragment is available.
 *
 * used as:
 * if ((1 << (size - 1)) & fragtbl[fs->fs_frag][map] {
 *      at least one fragment of the indicated size is available
 * }
 *
 * These tables are used by the scanc instruction on the VAX to
 * quickly find an appropriate fragment.
 */
const u_char fragtbl124[256] = {
        0x00, 0x16, 0x16, 0x2a, 0x16, 0x16, 0x26, 0x4e,
        0x16, 0x16, 0x16, 0x3e, 0x2a, 0x3e, 0x4e, 0x8a,
        0x16, 0x16, 0x16, 0x3e, 0x16, 0x16, 0x36, 0x5e,
        0x16, 0x16, 0x16, 0x3e, 0x3e, 0x3e, 0x5e, 0x9e,
        0x16, 0x16, 0x16, 0x3e, 0x16, 0x16, 0x36, 0x5e,
        0x16, 0x16, 0x16, 0x3e, 0x3e, 0x3e, 0x5e, 0x9e,
        0x2a, 0x3e, 0x3e, 0x2a, 0x3e, 0x3e, 0x2e, 0x6e,
        0x3e, 0x3e, 0x3e, 0x3e, 0x2a, 0x3e, 0x6e, 0xaa,
        0x16, 0x16, 0x16, 0x3e, 0x16, 0x16, 0x36, 0x5e,
        0x16, 0x16, 0x16, 0x3e, 0x3e, 0x3e, 0x5e, 0x9e,
        0x16, 0x16, 0x16, 0x3e, 0x16, 0x16, 0x36, 0x5e,
        0x16, 0x16, 0x16, 0x3e, 0x3e, 0x3e, 0x5e, 0x9e,
        0x26, 0x36, 0x36, 0x2e, 0x36, 0x36, 0x26, 0x6e,
        0x36, 0x36, 0x36, 0x3e, 0x2e, 0x3e, 0x6e, 0xae,
        0x4e, 0x5e, 0x5e, 0x6e, 0x5e, 0x5e, 0x6e, 0x4e,
        0x5e, 0x5e, 0x5e, 0x7e, 0x6e, 0x7e, 0x4e, 0xce,
        0x16, 0x16, 0x16, 0x3e, 0x16, 0x16, 0x36, 0x5e,
        0x16, 0x16, 0x16, 0x3e, 0x3e, 0x3e, 0x5e, 0x9e,
        0x16, 0x16, 0x16, 0x3e, 0x16, 0x16, 0x36, 0x5e,
        0x16, 0x16, 0x16, 0x3e, 0x3e, 0x3e, 0x5e, 0x9e,
        0x16, 0x16, 0x16, 0x3e, 0x16, 0x16, 0x36, 0x5e,
        0x16, 0x16, 0x16, 0x3e, 0x3e, 0x3e, 0x5e, 0x9e,
        0x3e, 0x3e, 0x3e, 0x3e, 0x3e, 0x3e, 0x3e, 0x7e,
        0x3e, 0x3e, 0x3e, 0x3e, 0x3e, 0x3e, 0x7e, 0xbe,
        0x2a, 0x3e, 0x3e, 0x2a, 0x3e, 0x3e, 0x2e, 0x6e,
        0x3e, 0x3e, 0x3e, 0x3e, 0x2a, 0x3e, 0x6e, 0xaa,
        0x3e, 0x3e, 0x3e, 0x3e, 0x3e, 0x3e, 0x3e, 0x7e,
        0x3e, 0x3e, 0x3e, 0x3e, 0x3e, 0x3e, 0x7e, 0xbe,
        0x4e, 0x5e, 0x5e, 0x6e, 0x5e, 0x5e, 0x6e, 0x4e,
        0x5e, 0x5e, 0x5e, 0x7e, 0x6e, 0x7e, 0x4e, 0xce,
        0x8a, 0x9e, 0x9e, 0xaa, 0x9e, 0x9e, 0xae, 0xce,
        0x9e, 0x9e, 0x9e, 0xbe, 0xaa, 0xbe, 0xce, 0x8a,
};

const u_char fragtbl8[256] = {
        0x00, 0x01, 0x01, 0x02, 0x01, 0x01, 0x02, 0x04,
        0x01, 0x01, 0x01, 0x03, 0x02, 0x03, 0x04, 0x08,
        0x01, 0x01, 0x01, 0x03, 0x01, 0x01, 0x03, 0x05,
        0x02, 0x03, 0x03, 0x02, 0x04, 0x05, 0x08, 0x10,
        0x01, 0x01, 0x01, 0x03, 0x01, 0x01, 0x03, 0x05,
        0x01, 0x01, 0x01, 0x03, 0x03, 0x03, 0x05, 0x09,
        0x02, 0x03, 0x03, 0x02, 0x03, 0x03, 0x02, 0x06,
        0x04, 0x05, 0x05, 0x06, 0x08, 0x09, 0x10, 0x20,
        0x01, 0x01, 0x01, 0x03, 0x01, 0x01, 0x03, 0x05,
        0x01, 0x01, 0x01, 0x03, 0x03, 0x03, 0x05, 0x09,
        0x01, 0x01, 0x01, 0x03, 0x01, 0x01, 0x03, 0x05,
        0x03, 0x03, 0x03, 0x03, 0x05, 0x05, 0x09, 0x11,
        0x02, 0x03, 0x03, 0x02, 0x03, 0x03, 0x02, 0x06,
        0x03, 0x03, 0x03, 0x03, 0x02, 0x03, 0x06, 0x0a,
        0x04, 0x05, 0x05, 0x06, 0x05, 0x05, 0x06, 0x04,
        0x08, 0x09, 0x09, 0x0a, 0x10, 0x11, 0x20, 0x40,
        0x01, 0x01, 0x01, 0x03, 0x01, 0x01, 0x03, 0x05,
        0x01, 0x01, 0x01, 0x03, 0x03, 0x03, 0x05, 0x09,
        0x01, 0x01, 0x01, 0x03, 0x01, 0x01, 0x03, 0x05,
        0x03, 0x03, 0x03, 0x03, 0x05, 0x05, 0x09, 0x11,
        0x01, 0x01, 0x01, 0x03, 0x01, 0x01, 0x03, 0x05,
        0x01, 0x01, 0x01, 0x03, 0x03, 0x03, 0x05, 0x09,
        0x03, 0x03, 0x03, 0x03, 0x03, 0x03, 0x03, 0x07,
        0x05, 0x05, 0x05, 0x07, 0x09, 0x09, 0x11, 0x21,
        0x02, 0x03, 0x03, 0x02, 0x03, 0x03, 0x02, 0x06,
        0x03, 0x03, 0x03, 0x03, 0x02, 0x03, 0x06, 0x0a,
        0x03, 0x03, 0x03, 0x03, 0x03, 0x03, 0x03, 0x07,
        0x02, 0x03, 0x03, 0x02, 0x06, 0x07, 0x0a, 0x12,
        0x04, 0x05, 0x05, 0x06, 0x05, 0x05, 0x06, 0x04,
        0x05, 0x05, 0x05, 0x07, 0x06, 0x07, 0x04, 0x0c,
        0x08, 0x09, 0x09, 0x0a, 0x09, 0x09, 0x0a, 0x0c,
        0x10, 0x11, 0x11, 0x12, 0x20, 0x21, 0x40, 0x80,
};

/*
 * The actual fragtbl array.
 */
const u_char *fragtbl[MAXFRAG + 1] = {
        NULL, fragtbl124, fragtbl124, NULL, fragtbl124, NULL, NULL, NULL,
        fragtbl8,
};
File: ffs_vfsops.c


/*      $OpenBSD: ffs_vfsops.c,v 1.188 2020/11/07 05:24:20 gnezdo Exp $     */
/*      $NetBSD: ffs_vfsops.c,v 1.19 1996/02/09 22:22:26 christos Exp $     */

/*
 * Copyright (c) 1989, 1991, 1993, 1994
 *      The Regents of the University of California.  All rights reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions
 * are met:
 * 1. Redistributions of source code must retain the above copyright
 *    notice, this list of conditions and the following disclaimer.
 * 2. Redistributions in binary form must reproduce the above copyright
 *    notice, this list of conditions and the following disclaimer in the
 *    documentation and/or other materials provided with the distribution.
 * 3. Neither the name of the University nor the names of its contributors
 *    may be used to endorse or promote products derived from this software
 *    without specific prior written permission.
 *
 * THIS SOFTWARE IS PROVIDED BY THE REGENTS AND CONTRIBUTORS ``AS IS'' AND
 * ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
 * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
 * ARE DISCLAIMED.  IN NO EVENT SHALL THE REGENTS OR CONTRIBUTORS BE LIABLE
 * FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
 * DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
 * OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
 * HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
 * LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
 * OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
 * SUCH DAMAGE.
 *
 *      @(#)ffs_vfsops.c 8.14 (Berkeley) 11/28/94
 */

#include <sys/param.h>
#include <sys/systm.h>
#include <sys/namei.h>
#include <sys/proc.h>
#include <sys/kernel.h>
#include <sys/vnode.h>
#include <sys/socket.h>
#include <sys/mount.h>
#include <sys/buf.h>
#include <sys/mbuf.h>
#include <sys/fcntl.h>
#include <sys/ioctl.h>
#include <sys/errno.h>
#include <sys/malloc.h>
#include <sys/sysctl.h>
#include <sys/pool.h>
#include <sys/dkio.h>
#include <sys/disk.h>
#include <sys/specdev.h>

#include <ufs/ufs/quota.h>
#include <ufs/ufs/ufsmount.h>
#include <ufs/ufs/inode.h>
```

```c
#include <ufs/ufs/dir.h>
#include <ufs/ufs/ufs_extern.h>
#include <ufs/ufs/dirhash.h>

#include <ufs/ffs/fs.h>
#include <ufs/ffs/ffs_extern.h>

#include <uvm/uvm_extern.h>

int ffs_sbupdate(struct ufsmount *, int);
int ffs_reload_vnode(struct vnode *, void *);
int ffs_sync_vnode(struct vnode *, void *);
int ffs_validate(struct fs *);

void ffs1_compat_read(struct fs *, struct ufsmount *, daddr_t);
void ffs1_compat_write(struct fs *, struct ufsmount *);

const struct vfsops ffs_vfsops = {
        .vfs_mount      = ffs_mount,
        .vfs_start      = ufs_start,
        .vfs_unmount    = ffs_unmount,
        .vfs_root       = ufs_root,
        .vfs_quotactl   = ufs_quotactl,
        .vfs_statfs     = ffs_statfs,
        .vfs_sync       = ffs_sync,
        .vfs_vget       = ffs_vget,
        .vfs_fhtovp     = ffs_fhtovp,
        .vfs_vptofh     = ffs_vptofh,
        .vfs_init       = ffs_init,
        .vfs_sysctl     = ffs_sysctl,
        .vfs_checkexp   = ufs_check_export,
};

struct inode_vtbl ffs_vtbl = {
        ffs_truncate,
        ffs_update,
        ffs_inode_alloc,
        ffs_inode_free,
        ffs_balloc,
        ffs_bufatoff
};

int
ffs_checkrange(struct mount *mp, uint32_t ino)
{
        struct buf *bp;
        struct cg *cgp;
        struct fs *fs;
        struct ufsmount *ump;
        int cg, error;

        fs = VFSTOUFS(mp)->um_fs;
        if (ino < ROOTINO || ino >= fs->fs_ncg * fs->fs_ipg)
                return ESTALE;

        /*
         * Need to check if inode is initialized because ffsv2 does
         * lazy initialization and we can get here from nfs_fhtovp
         */
        if (fs->fs_magic != FS_UFS2_MAGIC)
                return 0;

        cg = ino_to_cg(fs, ino);
        ump = VFSTOUFS(mp);

        error = bread(ump->um_devvp, fsbtodb(fs, cgtod(fs, cg)),
            (int)fs->fs_cgsize, &bp);
        if (error)
                return error;

        cgp = (struct cg *)bp->b_data;
        if (!cg_chkmagic(cgp)) {
                brelse(bp);
                return ESTALE;
        }

        brelse(bp);

        if (cg * fs->fs_ipg + cgp->cg_initediblk < ino)
                return ESTALE;

        return 0;
}

/*
 * Called by main() when ufs is going to be mounted as root.
 */

struct pool ffs_ino_pool;
struct pool ffs_dinode1_pool;
#ifdef FFS2
struct pool ffs_dinode2_pool;
#endif

int
ffs_mountroot(void)
{
        struct fs *fs;
        struct mount *mp;
        struct proc *p = curproc; /* XXX */
        struct ufsmount *ump;
        int error;

        /*
         * Get vnodes for swapdev and rootdev.
         */
        swapdev_vp = NULL;
        if ((error = bdevvp(swapdev, &swapdev_vp)) ||
            (error = bdevvp(rootdev, &rootvp))) {
                printf("ffs_mountroot: can't setup bdevvp's\n");
                if (swapdev_vp)
                        vrele(swapdev_vp);
                return (error);
        }

        if ((error = vfs_rootmountalloc("ffs", "root_device", &mp)) != 0) {
                vrele(swapdev_vp);
                vrele(rootvp);
                return (error);
        }

        if ((error = ffs_mountfs(rootvp, mp, p)) != 0) {
                vfs_unbusy(mp);
                vfs_mount_free(mp);
                vrele(swapdev_vp);
                vrele(rootvp);
                return (error);
        }

        TAILQ_INSERT_TAIL(&mountlist, mp, mnt_list);
        ump = VFSTOUFS(mp);
        fs = ump->um_fs;
        strlcpy(fs->fs_fsmnt, mp->mnt_stat.f_mntonname, sizeof(fs->fs_fsmnt));
        (void)ffs_statfs(mp, &mp->mnt_stat, p);
        vfs_unbusy(mp);
        inittodr(fs->fs_time);

        return (0);
}

/*
 * VFS Operations.
 *
 * mount system call
 */
int
ffs_mount(struct mount *mp, const char *path, void *data,
    struct nameidata *ndp, struct proc *p)
{
        struct vnode *devvp;
        struct ufs_args *args = data;
        struct ufsmount *ump = NULL;
        struct fs *fs;
        char fname[MNAMELEN];
        char fspec[MNAMELEN];
        int error = 0, flags;
        int ronly;

#ifndef FFS_SOFTUPDATES
        if (mp->mnt_flag & MNT_SOFTDEP) {
                printf("WARNING: soft updates isn't compiled in\n");
                mp->mnt_flag &= ~MNT_SOFTDEP;
        }
#endif

        /*
         * Soft updates is incompatible with "async",
         * so if we are doing softupdates stop the user
         * from setting the async flag.
         */
        if ((mp->mnt_flag & (MNT_SOFTDEP | MNT_ASYNC)) ==
            (MNT_SOFTDEP | MNT_ASYNC)) {
                return (EINVAL);
        }
        /*
         * If updating, check whether changing from read-only to
         * read/write; if there is no device name, that's all we do.
         */
        if (mp->mnt_flag & MNT_UPDATE) {
                ump = VFSTOUFS(mp);
                fs = ump->um_fs;
                devvp = ump->um_devvp;
                error = 0;
                ronly = fs->fs_ronly;

                /*
                 * Soft updates won't be set if read/write,
                 * so "async" will be illegal.
                 */
                if (ronly == 0 && (mp->mnt_flag & MNT_ASYNC) &&
                    (fs->fs_flags & FS_DOSOFTDEP)) {
                        error = EINVAL;
                        goto error_1;
                }

                if (ronly == 0 && (mp->mnt_flag & MNT_RDONLY)) {
                        /* Flush any dirty data */
                        VFS_SYNC(mp, MNT_WAIT, 0, p->p_ucred, p);

                        /*
                         * Get rid of files open for writing.
                         */
                        flags = WRITECLOSE;
                        if (args == NULL)
                                flags |= IGNORECLEAN;
                        if (mp->mnt_flag & MNT_FORCE)
                                flags |= FORCECLOSE;
                        if (fs->fs_flags & FS_DOSOFTDEP) {
                                error = softdep_flushfiles(mp, flags, p);
                                mp->mnt_flag &= ~MNT_SOFTDEP;
                        } else
                                error = ffs_flushfiles(mp, flags, p);
                        mp->mnt_flag |= MNT_RDONLY;
                        ronly = 1;
                }

                /*
                 * Flush soft dependencies if disabling it via an update
                 * mount. This may leave some items to be processed,
                 * so don't do this yet XXX.
                 */
                if ((fs->fs_flags & FS_DOSOFTDEP) &&
                    !(mp->mnt_flag & MNT_SOFTDEP) &&
                    !(mp->mnt_flag & MNT_RDONLY) && fs->fs_ronly == 0) {
#if 0
                        flags = WRITECLOSE;
                        if (mp->mnt_flag & MNT_FORCE)
                                flags |= FORCECLOSE;
                        error = softdep_flushfiles(mp, flags, p);
#elif FFS_SOFTUPDATES
                        mp->mnt_flag |= MNT_SOFTDEP;
#endif
                }
                /*
                 * When upgrading to a softdep mount, we must first flush
                 * all vnodes. (not done yet -- see above)
                 */
                if (!(fs->fs_flags & FS_DOSOFTDEP) &&
                    (mp->mnt_flag & MNT_SOFTDEP) && fs->fs_ronly == 0) {
#if 0
                        flags = WRITECLOSE;
                        if (mp->mnt_flag & MNT_FORCE)
                                flags |= FORCECLOSE;
                        error = ffs_flushfiles(mp, flags, p);
#else
                        mp->mnt_flag &= ~MNT_SOFTDEP;
#endif
                }

                if (!error && (mp->mnt_flag & MNT_RELOAD))
                        error = ffs_reload(mp, ndp->ni_cnd.cn_cred, p);
                if (error)
                        goto error_1;

                if (ronly && (mp->mnt_flag & MNT_WANTRDWR)) {
                        if (fs->fs_clean == 0) {
#if 0
                                /*
                                 * It is safe to mount an unclean file system
                                 * if it was previously mounted with softdep
                                 * but we may lose space and must
                                 * sometimes run fsck manually.
                                 */
                                if (fs->fs_flags & FS_DOSOFTDEP)
                                        printf(
"WARNING: %s was not properly unmounted\n",
                                            fs->fs_fsmnt);
                                else
#endif
                                if (mp->mnt_flag & MNT_FORCE) {
                                        printf(
"WARNING: %s was not properly unmounted\n",
                                            fs->fs_fsmnt);
                                } else {
                                        printf(
"WARNING: R/W mount of %s denied.  Filesystem is not clean - run fsck\n",
                                            fs->fs_fsmnt);
                                        error = EROFS;
                                        goto error_1;
                                }
                        }

                        if ((fs->fs_flags & FS_DOSOFTDEP)) {
                                error = softdep_mount(devvp, mp, fs,
                                                  p->p_ucred);
                                if (error)
                                        goto error_1;
                        }
                        fs->fs_contigdirs = malloc((u_long)fs->fs_ncg,
                             M_UFSMNT, M_WAITOK|M_ZERO);

                        ronly = 0;
                }
                if (args == NULL)
                        goto success;
                if (args->fspec == NULL) {
                        /*
                         * Process export requests.
                         */
                        error = vfs_export(mp, &ump->um_export,
                            &args->export_info);
                        if (error)
                                goto error_1;
                        else
```

```c
                                goto success;
                }
        }

        /*
         * Not an update, or updating the name: look up the name
         * and verify that it refers to a sensible block device.
         */
        error = copyinstr(args->fspec, fspec, sizeof(fspec), NULL);
        if (error)
                goto error_1;

        if (disk_map(fspec, fname, MNAMELEN, DM_OPENBLCK) == -1)
                memcpy(fname, fspec, sizeof(fname));

        NDINIT(ndp, LOOKUP, FOLLOW, UIO_SYSSPACE, fname, p);
        if ((error = namei(ndp)) != 0)
                goto error_1;

        devvp = ndp->ni_vp;

        if (devvp->v_type != VBLK) {
                error = ENOTBLK;
                goto error_2;
        }

        if (major(devvp->v_rdev) >= nblkdev) {
                error = ENXIO;
                goto error_2;
        }

        if (mp->mnt_flag & MNT_UPDATE) {
                /*
                 * UPDATE
                 * If it's not the same vnode, or at least the same device
                 * then it's not correct.
                 */

                if (devvp != ump->um_devvp) {
                        if (devvp->v_rdev == ump->um_devvp->v_rdev) {
                                vrele(devvp);
                        } else {
                                error = EINVAL;  /* needs translation */
                        }
                } else
                        vrele(devvp);
                /*
                 * Update device name only on success
                 */
                if (!error) {
                        /*
                         * Save "mounted from" info for mount point (NULL pad)
                         */
                        memset(mp->mnt_stat.f_mntfromname, 0, MNAMELEN);
                        strlcpy(mp->mnt_stat.f_mntfromname, fname, MNAMELEN);
                        memset(mp->mnt_stat.f_mntfromspec, 0, MNAMELEN);
                        strlcpy(mp->mnt_stat.f_mntfromspec, fspec, MNAMELEN);
                }
        } else {
                /*
                 * Since this is a new mount, we want the names for
                 * the device and the mount point copied in.  If an
                 * error occurs,  the mountpoint is discarded by the
                 * upper level code.
                 */
                memset(mp->mnt_stat.f_mntonname, 0, MNAMELEN);
                strlcpy(mp->mnt_stat.f_mntonname, path, MNAMELEN);
                memset(mp->mnt_stat.f_mntfromname, 0, MNAMELEN);
                strlcpy(mp->mnt_stat.f_mntfromname, fname, MNAMELEN);
                memset(mp->mnt_stat.f_mntfromspec, 0, MNAMELEN);
                strlcpy(mp->mnt_stat.f_mntfromspec, fspec, MNAMELEN);

                error = ffs_mountfs(devvp, mp, p);
        }

        if (error)
                goto error_2;

        /*
         * Initialize FS stat information in mount struct; uses both
         * mp->mnt_stat.f_mntonname and mp->mnt_stat.f_mntfromname
         *
         * This code is common to root and non-root mounts
         */
        if (args)
                memcpy(&mp->mnt_stat.mount_info.ufs_args, args, sizeof(*args));
        VFS_STATFS(mp, &mp->mnt_stat, p);

success:
        if (path && (mp->mnt_flag & MNT_UPDATE)) {
                /* Update clean flag after changing read-onlyness. */
                fs = ump->um_fs;
                if (ronly != fs->fs_ronly) {
                        fs->fs_ronly = ronly;
                        fs->fs_clean = ronly &&
                            (fs->fs_flags & FS_UNCLEAN) == 0 ? 1 : 0;
                        if (ronly)
                                free(fs->fs_contigdirs, M_UFSMNT, fs->fs_ncg);
                }
                if (!ronly) {
                        if (mp->mnt_flag & MNT_SOFTDEP)
                                fs->fs_flags |= FS_DOSOFTDEP;
                        else
                                fs->fs_flags &= ~FS_DOSOFTDEP;
                }
                ffs_sbupdate(ump, MNT_WAIT);
#if 0
                if (ronly) {
                        int force = 0;

                        /*
                         * Updating mount to readonly. Try a cache flush.
                         * Ignore error because the ioctl may not be supported.
                         */
                        VOP_IOCTL(ump->um_devvp, DIOCCACHESYNC, &force,
                            FWRITE, FSCRED, p);
              }
#endif
        }
        return (0);

error_2:/* error with devvp held */
        vrele (devvp);

error_1:/* no state to back out */
        return (error);
}

struct ffs_reload_args {
        struct fs *fs;
        struct proc *p;
        struct ucred *cred;
        struct vnode *devvp;
};

int
ffs_reload_vnode(struct vnode *vp, void *args)
{
        struct ffs_reload_args *fra = args;
        struct inode *ip;
        struct buf *bp;
        int error;

        /*
         * Step 4: invalidate all inactive vnodes.
         */
        if (vp->v_usecount == 0) {
                vgonel(vp, fra->p);
                return (0);
        }

        /*
         * Step 5: invalidate all cached file data.
         */
        if (vget(vp, LK_EXCLUSIVE))
                return (0);

        if (vinvalbuf(vp, 0, fra->cred, fra->p, 0, INFSLP))
                panic("ffs_reload: dirty2");

        /*
         * Step 6: re-read inode data for all active vnodes.
         */
        ip = VTOI(vp);

        error = bread(fra->devvp,
            fsbtodb(fra->fs, ino_to_fsba(fra->fs, ip->i_number)),
            (int)fra->fs->fs_bsize, &bp);
        if (error) {
                brelse(bp);
                vput(vp);
                return (error);
        }

        if (fra->fs->fs_magic == FS_UFS1_MAGIC)
                *ip->i_din1 = *((struct ufs1_dinode *)bp->b_data +
                    ino_to_fsbo(fra->fs, ip->i_number));
#ifdef FFS2
        else
                *ip->i_din2 = *((struct ufs2_dinode *)bp->b_data +
                    ino_to_fsbo(fra->fs, ip->i_number));
#endif /* FFS2 */
        ip->i_effnlink = DIP(ip, nlink);
        brelse(bp);
        vput(vp);
        return (0);
}

/*
 * Reload all incore data for a filesystem (used after running fsck on
 * the root filesystem and finding things to fix). The filesystem must
 * be mounted read-only.
 *
 * Things to do to update the mount:
 *      1) invalidate all cached meta-data.
 *      2) re-read superblock from disk.
 *      3) re-read summary information from disk.
 *      4) invalidate all inactive vnodes.
 *      5) invalidate all cached file data.
 *      6) re-read inode data for all active vnodes.
 */
int
ffs_reload(struct mount *mountp, struct ucred *cred, struct proc *p)
{
        struct vnode *devvp;
        caddr_t space;
        struct fs *fs, *newfs;
        int i, blks, size, error;
        int32_t *lp;
        struct buf *bp = NULL;
        struct ffs_reload_args fra;

        if ((mountp->mnt_flag & MNT_RDONLY) == 0)
                return (EINVAL);
        /*
         * Step 1: invalidate all cached meta-data.
         */
        devvp = VFSTOUFS(mountp)->um_devvp;
        vn_lock(devvp, LK_EXCLUSIVE | LK_RETRY);
        error = vinvalbuf(devvp, 0, cred, p, 0, INFSLP);
        VOP_UNLOCK(devvp);
        if (error)
                panic("ffs_reload: dirty1");

        /*
         * Step 2: re-read superblock from disk.
         */
        fs = VFSTOUFS(mountp)->um_fs;

        error = bread(devvp, fs->fs_sblockloc / DEV_BSIZE, SBSIZE, &bp);
        if (error) {
                brelse(bp);
                return (error);
        }

        newfs = (struct fs *)bp->b_data;
        if (ffs_validate(newfs) == 0) {
                brelse(bp);
                return (EINVAL);
        }

        /*
         * Copy pointer fields back into superblock before copying in       XXX
         * new superblock. These should really be in the ufsmount. XXX
         * Note that important parameters (eg fs_ncg) are unchanged.
         */
        newfs->fs_csp = fs->fs_csp;
        newfs->fs_maxcluster = fs->fs_maxcluster;
        newfs->fs_ronly = fs->fs_ronly;
        memcpy(fs, newfs, fs->fs_sbsize);
        if (fs->fs_sbsize < SBSIZE)
                bp->b_flags |= B_INVAL;
        brelse(bp);
        VFSTOUFS(mountp)->um_maxsymlinklen = fs->fs_maxsymlinklen;
        ffs1_compat_read(fs, VFSTOUFS(mountp), fs->fs_sblockloc);
        ffs_oldfscompat(fs);
        (void)ffs_statfs(mountp, &mountp->mnt_stat, p);
        /*
         * Step 3: re-read summary information from disk.
         */
        blks = howmany(fs->fs_cssize, fs->fs_fsize);
        space = (caddr_t)fs->fs_csp;
        for (i = 0; i < blks; i += fs->fs_frag) {
                size = fs->fs_bsize;
                if (i + fs->fs_frag > blks)
                        size = (blks - i) * fs->fs_fsize;
                error = bread(devvp, fsbtodb(fs, fs->fs_csaddr + i), size, &bp);
                if (error) {
                        brelse(bp);
                        return (error);
                }
                memcpy(space, bp->b_data, size);
                space += size;
                brelse(bp);
        }
        if ((fs->fs_flags & FS_DOSOFTDEP))
                (void) softdep_mount(devvp, mountp, fs, cred);
        /*
         * We no longer know anything about clusters per cylinder group.
         */
        if (fs->fs_contigsumsize > 0) {
                lp = fs->fs_maxcluster;
                for (i = 0; i < fs->fs_ncg; i++)
                        *lp++ = fs->fs_contigsumsize;
        }

        fra.p = p;
        fra.cred = cred;
        fra.fs = fs;
        fra.devvp = devvp;

        error = vfs_mount_foreach_vnode(mountp, ffs_reload_vnode, &fra);

        return (error);
}

/*
 * Checks if a super block is sane enough to be mounted.
 */
int
ffs_validate(struct fs *fsp)
{
#ifdef FFS2
        if (fsp->fs_magic != FS_UFS2_MAGIC && fsp->fs_magic != FS_UFS1_MAGIC)
```

```
                return (0); /* Invalid magic */
#else
        if (fsp->fs_magic != FS_UFS1_MAGIC)
                return (0); /* Invalid magic */
#endif /* FFS2 */

        if ((u_int)fsp->fs_bsize > MAXBSIZE)
                return (0); /* Invalid block size */

        if ((u_int)fsp->fs_bsize < sizeof(struct fs))
                return (0); /* Invalid block size */

        if ((u_int)fsp->fs_sbsize > SBSIZE)
                return (0); /* Invalid super block size */

        if ((u_int)fsp->fs_frag > MAXFRAG || fragtbl[fsp->fs_frag] == NULL)
                return (0); /* Invalid number of fragments */

        if (fsp->fs_inodefmt == FS_42INODEFMT)
                fsp->fs_maxsymlinklen = 0;
        else if (fsp->fs_maxsymlinklen < 0)
                return (0); /* Invalid max size of short symlink */

        return (1); /* Super block is okay */
}

/*
 * Possible locations for the super-block.
 */
const int sbtry[] = SBLOCKSEARCH;

/*
 * Common code for mount and mountroot
 */
int
ffs_mountfs(struct vnode *devvp, struct mount *mp, struct proc *p)
{
        struct ufsmount *ump;
        struct buf *bp;
        struct fs *fs;
        dev_t dev;
        caddr_t space;
        daddr_t sbloc;
        int error, i, blks, size, ronly;
        int32_t *lp;
        struct ucred *cred;
        u_int64_t maxfilesize;                                  /* XXX */

        dev = devvp->v_rdev;
        cred = p ? p->p_ucred : NOCRED;
        /*
         * Disallow multiple mounts of the same device.
         * Disallow mounting of a device that is currently in use
         * (except for root, which might share swap device for miniroot).
         * Flush out any old buffers remaining from a previous use.
         */
        if ((error = vfs_mountedon(devvp)) != 0)
                return (error);
        if (vcount(devvp) > 1 && devvp != rootvp)
                return (EBUSY);
        vn_lock(devvp, LK_EXCLUSIVE | LK_RETRY);
        error = vinvalbuf(devvp, V_SAVE, cred, p, 0, INFSLP);
        VOP_UNLOCK(devvp);
        if (error)
                return (error);

        ronly = (mp->mnt_flag & MNT_RDONLY) != 0;
        error = VOP_OPEN(devvp, ronly ? FREAD : FREAD|FWRITE, FSCRED, p);
        if (error)
                return (error);

        bp = NULL;
        ump = NULL;

        /*
         * Try reading the super-block in each of its possible locations.
         */
        for (i = 0; sbtry[i] != -1; i++) {
                if (bp != NULL) {
                        bp->b_flags |= B_NOCACHE;
                        brelse(bp);
                        bp = NULL;
                }

                error = bread(devvp, sbtry[i] / DEV_BSIZE, SBSIZE, &bp);
                if (error)
                        goto out;

                fs = (struct fs *) bp->b_data;
                sbloc = sbtry[i];

                /*
                 * Do not look for an FFS1 file system at SBLOCK_UFS2. Doing so
                 * will find the wrong super-block for file systems with 64k
                 * block size.
                 */
                if (fs->fs_magic == FS_UFS1_MAGIC && sbloc == SBLOCK_UFS2)
                        continue;

                if (ffs_validate(fs))
                        break; /* Super block validated */
        }

        if (sbtry[i] == -1) {
                error = EINVAL;
                goto out;
        }

        fs->fs_fmod = 0;
        fs->fs_flags &= ~FS_UNCLEAN;
        if (fs->fs_clean == 0) {
#if 0
                /*
                 * It is safe to mount an unclean file system
                 * if it was previously mounted with softdep
                 * but we may lose space and must
                 * sometimes run fsck manually.
                 */
                if (fs->fs_flags & FS_DOSOFTDEP)
                        printf(
"WARNING: %s was not properly unmounted\n",
                            fs->fs_fsmnt);
                else
#endif
                if (ronly || (mp->mnt_flag & MNT_FORCE)) {
                        printf(
"WARNING: %s was not properly unmounted\n",
                            fs->fs_fsmnt);
                } else {
                        printf(
"WARNING: R/W mount of %s denied.  Filesystem is not clean - run fsck\n",
                            fs->fs_fsmnt);
                        error = EROFS;
                        goto out;
                }
        }

        if (fs->fs_postblformat == FS_42POSTBLFMT && !ronly) {
#ifndef SMALL_KERNEL
                printf("ffs_mountfs(): obsolete rotational table format, "
                    "please use fsck_ffs(8) -c 1\n");
#endif
                error = EROFS;
                goto out;
        }

        ump = malloc(sizeof *ump, M_UFSMNT, M_WAITOK|M_ZERO);
        ump->um_fs = malloc((u_long)fs->fs_sbsize, M_UFSMNT,
            M_WAITOK);

        if (fs->fs_magic == FS_UFS1_MAGIC)
                ump->um_fstype = UM_UFS1;
#ifdef FFS2
        else
                ump->um_fstype = UM_UFS2;
#endif

        memcpy(ump->um_fs, bp->b_data, fs->fs_sbsize);
        if (fs->fs_sbsize < SBSIZE)
                bp->b_flags |= B_INVAL;
        brelse(bp);
        bp = NULL;
        fs = ump->um_fs;

        ffs1_compat_read(fs, ump, sbloc);

        if (fs->fs_clean == 0)
                fs->fs_flags |= FS_UNCLEAN;
        fs->fs_ronly = ronly;
        size = fs->fs_cssize;
        blks = howmany(size, fs->fs_fsize);
        if (fs->fs_contigsumsize > 0)
                size += fs->fs_ncg * sizeof(int32_t);
        space = malloc((u_long)size, M_UFSMNT, M_WAITOK);
        fs->fs_csp = (struct csum *)space;
        for (i = 0; i < blks; i += fs->fs_frag) {
                size = fs->fs_bsize;
                if (i + fs->fs_frag > blks)
                        size = (blks - i) * fs->fs_fsize;
                error = bread(devvp, fsbtodb(fs, fs->fs_csaddr + i), size, &bp);
                if (error) {
                        free(fs->fs_csp, M_UFSMNT, 0);
                        goto out;
                }
                memcpy(space, bp->b_data, size);
                space += size;
                brelse(bp);
                bp = NULL;
        }
        if (fs->fs_contigsumsize > 0) {
                fs->fs_maxcluster = lp = (int32_t *)space;
                for (i = 0; i < fs->fs_ncg; i++)
                        *lp++ = fs->fs_contigsumsize;
        }
        mp->mnt_data = ump;
        mp->mnt_stat.f_fsid.val[0] = (long)dev;
        /* Use on-disk fsid if it exists, else fake it */
        if (fs->fs_id[0] != 0 && fs->fs_id[1] != 0)
                mp->mnt_stat.f_fsid.val[1] = fs->fs_id[1];
        else
                mp->mnt_stat.f_fsid.val[1] = mp->mnt_vfc->vfc_typenum;
        mp->mnt_stat.f_namemax = MAXNAMLEN;
        mp->mnt_flag |= MNT_LOCAL;
        ump->um_mountp = mp;
        ump->um_dev = dev;
        ump->um_devvp = devvp;
        ump->um_nindir = fs->fs_nindir;
        ump->um_bptrtodb = fs->fs_fsbtodb;
        ump->um_seqinc = fs->fs_frag;
        ump->um_maxsymlinklen = fs->fs_maxsymlinklen;
        for (i = 0; i < MAXQUOTAS; i++)
                ump->um_quotas[i] = NULLVP;

        devvp->v_specmountpoint = mp;
        ffs_oldfscompat(fs);

        if (ronly)
                fs->fs_contigdirs = NULL;
        else {
                fs->fs_contigdirs = malloc((u_long)fs->fs_ncg,
                    M_UFSMNT, M_WAITOK|M_ZERO);
        }

        /*
         * Set FS local "last mounted on" information (NULL pad)
         */
        memset(fs->fs_fsmnt, 0, sizeof(fs->fs_fsmnt));
        strlcpy(fs->fs_fsmnt, mp->mnt_stat.f_mntonname, sizeof(fs->fs_fsmnt));

#if 0
        if( mp->mnt_flag & MNT_ROOTFS) {
                /*
                 * Root mount; update timestamp in mount structure.
                 * this will be used by the common root mount code
                 * to update the system clock.
                 */
                mp->mnt_time = fs->fs_time;
        }
#endif

        /*
         * XXX
         * Limit max file size.  Even though ffs can handle files up to 16TB,
         * we do limit the max file to 2^31 pages to prevent overflow of
         * a 32-bit unsigned int.  The buffer cache has its own checks but
         * a little added paranoia never hurts.
         */
        ump->um_savedmaxfilesize = fs->fs_maxfilesize;          /* XXX */
        maxfilesize = FS_KERNMAXFILESIZE(PAGE_SIZE, fs);
        if (fs->fs_maxfilesize > maxfilesize)                   /* XXX */
                fs->fs_maxfilesize = maxfilesize;       /* XXX */
        if (ronly == 0) {
                if ((fs->fs_flags & FS_DOSOFTDEP) &&
                    (error = softdep_mount(devvp, mp, fs, cred)) != 0) {
                        free(fs->fs_csp, M_UFSMNT, 0);
                        free(fs->fs_contigdirs, M_UFSMNT, fs->fs_ncg);
                        goto out;
                }
                fs->fs_fmod = 1;
                fs->fs_clean = 0;
                if (mp->mnt_flag & MNT_SOFTDEP)
                        fs->fs_flags |= FS_DOSOFTDEP;
                else
                        fs->fs_flags &= ~FS_DOSOFTDEP;
                error = ffs_sbupdate(ump, MNT_WAIT);
                if (error == EROFS)
                        goto out;
        }
        return (0);
out:
        if (devvp->v_specinfo)
                devvp->v_specmountpoint = NULL;
        if (bp)
                brelse(bp);

        vn_lock(devvp, LK_EXCLUSIVE|LK_RETRY);
        (void)VOP_CLOSE(devvp, ronly ? FREAD : FREAD|FWRITE, cred, p);
        VOP_UNLOCK(devvp);

        if (ump) {
                free(ump->um_fs, M_UFSMNT, ump->um_fs->fs_sbsize);
                free(ump, M_UFSMNT, sizeof(*ump));
                mp->mnt_data = NULL;
        }
        return (error);
}

/*
 * Sanity checks for old file systems.
 */
int
ffs_oldfscompat(struct fs *fs)
{
        int i;

        fs->fs_npsect = max(fs->fs_npsect, fs->fs_nsect);  /* XXX */
        fs->fs_interleave = max(fs->fs_interleave, 1);          /* XXX */
        if (fs->fs_postblformat == FS_42POSTBLFMT)      /* XXX */
                fs->fs_nrpos = 8;                               /* XXX */
        if (fs->fs_inodefmt < FS_44INODEFMT) {                  /* XXX */
                u_int64_t sizepb = fs->fs_bsize;        /* XXX */
                                                                /* XXX */
                fs->fs_maxfilesize = fs->fs_bsize * NDADDR - 1;  /* XXX */
                for (i = 0; i < NIADDR; i++) {                  /* XXX */
```

```c
                        sizepb *= NINDIR(fs);                   /* XXX */
                        fs->fs_maxfilesize += sizepb;           /* XXX */
                }                                               /* XXX */
                fs->fs_qbmask = ~fs->fs_bmask;                  /* XXX */
                fs->fs_qfmask = ~fs->fs_fmask;                  /* XXX */
        }                                                       /* XXX */
        if (fs->fs_avgfilesize <= 0)                            /* XXX */
                fs->fs_avgfilesize = AVFILESIZ;                 /* XXX */
        if (fs->fs_avgfpdir <= 0)                       /* XXX */
                fs->fs_avgfpdir = AFPDIR;               /* XXX */
        return (0);
}

/*
 * Auxiliary function for reading FFS1 super blocks.
 */
void
ffs1_compat_read(struct fs *fs, struct ufsmount *ump, daddr_t sbloc)
{
        if (fs->fs_magic == FS_UFS2_MAGIC)
                return; /* UFS2 */
#if 0
        if (fs->fs_ffs1_flags & FS_FLAGS_UPDATED)
                return; /* Already updated */
#endif
        fs->fs_flags = fs->fs_ffs1_flags;
        fs->fs_sblockloc = sbloc;
        fs->fs_maxbsize = fs->fs_bsize;
        fs->fs_time = fs->fs_ffs1_time;
        fs->fs_size = fs->fs_ffs1_size;
        fs->fs_dsize = fs->fs_ffs1_dsize;
        fs->fs_csaddr = fs->fs_ffs1_csaddr;
        fs->fs_cstotal.cs_ndir = fs->fs_ffs1_cstotal.cs_ndir;
        fs->fs_cstotal.cs_nbfree = fs->fs_ffs1_cstotal.cs_nbfree;
        fs->fs_cstotal.cs_nifree = fs->fs_ffs1_cstotal.cs_nifree;
        fs->fs_cstotal.cs_nffree = fs->fs_ffs1_cstotal.cs_nffree;
        fs->fs_ffs1_flags |= FS_FLAGS_UPDATED;
}

/*
 * Auxiliary function for writing FFS1 super blocks.
 */
void
ffs1_compat_write(struct fs *fs, struct ufsmount *ump)
{
        if (fs->fs_magic != FS_UFS1_MAGIC)
                return; /* UFS2 */

        fs->fs_ffs1_time = fs->fs_time;
        fs->fs_ffs1_cstotal.cs_ndir = fs->fs_cstotal.cs_ndir;
        fs->fs_ffs1_cstotal.cs_nbfree = fs->fs_cstotal.cs_nbfree;
        fs->fs_ffs1_cstotal.cs_nifree = fs->fs_cstotal.cs_nifree;
        fs->fs_ffs1_cstotal.cs_nffree = fs->fs_cstotal.cs_nffree;
}

/*
 * unmount system call
 */
int
ffs_unmount(struct mount *mp, int mntflags, struct proc *p)
{
        struct ufsmount *ump;
        struct fs *fs;
        int error, flags;

        flags = 0;
        if (mntflags & MNT_FORCE)
                flags |= FORCECLOSE;

        ump = VFSTOUFS(mp);
        fs = ump->um_fs;
        if (mp->mnt_flag & MNT_SOFTDEP)
                error = softdep_flushfiles(mp, flags, p);
        else
                error = ffs_flushfiles(mp, flags, p);
        if (error != 0)
                return (error);

        if (fs->fs_ronly == 0) {
                fs->fs_clean = (fs->fs_flags & FS_UNCLEAN) ? 0 : 1;
                error = ffs_sbupdate(ump, MNT_WAIT);
                /* ignore write errors if mounted RW on read-only device */
                if (error && error != EROFS) {
                        fs->fs_clean = 0;
                        return (error);
                }
                free(fs->fs_contigdirs, M_UFSMNT, fs->fs_ncg);
        }
        ump->um_devvp->v_specmountpoint = NULL;

        vn_lock(ump->um_devvp, LK_EXCLUSIVE | LK_RETRY);
        vinvalbuf(ump->um_devvp, V_SAVE, NOCRED, p, 0, INFSLP);
        (void)VOP_CLOSE(ump->um_devvp, fs->fs_ronly ? FREAD : FREAD|FWRITE,
            NOCRED, p);
        vput(ump->um_devvp);
        free(fs->fs_csp, M_UFSMNT, 0);
        free(fs, M_UFSMNT, fs->fs_sbsize);
        free(ump, M_UFSMNT, sizeof(*ump));
        mp->mnt_data = NULL;
        mp->mnt_flag &= ~MNT_LOCAL;
        return (0);
}

/*
 * Flush out all the files in a filesystem.
 */
int
ffs_flushfiles(struct mount *mp, int flags, struct proc *p)
{
        struct ufsmount *ump;
        int error;

        ump = VFSTOUFS(mp);
        if (mp->mnt_flag & MNT_QUOTA) {
                int i;
                if ((error = vflush(mp, NULLVP, SKIPSYSTEM|flags)) != 0)
                        return (error);
                for (i = 0; i < MAXQUOTAS; i++) {
                        if (ump->um_quotas[i] == NULLVP)
                                continue;
                        quotaoff(p, mp, i);
                }
                /*
                 * Here we fall through to vflush again to ensure
                 * that we have gotten rid of all the system vnodes.
                 */
        }

        /*
         * Flush all the files.
         */
        if ((error = vflush(mp, NULL, flags)) != 0)
                return (error);
        /*
         * Flush filesystem metadata.
         */
        vn_lock(ump->um_devvp, LK_EXCLUSIVE | LK_RETRY);
        error = VOP_FSYNC(ump->um_devvp, p->p_ucred, MNT_WAIT, p);
        VOP_UNLOCK(ump->um_devvp);
        return (error);
}

/*
 * Get file system statistics.
 */
int
ffs_statfs(struct mount *mp, struct statfs *sbp, struct proc *p)
{
        struct ufsmount *ump;
        struct fs *fs;

        ump = VFSTOUFS(mp);
        fs = ump->um_fs;

#ifdef FFS2
        if (fs->fs_magic != FS_MAGIC && fs->fs_magic != FS_UFS2_MAGIC)
                panic("ffs_statfs");
#else
        if (fs->fs_magic != FS_MAGIC)
                panic("ffs_statfs");
#endif /* FFS2 */

        sbp->f_bsize = fs->fs_fsize;
        sbp->f_iosize = fs->fs_bsize;
        sbp->f_blocks = fs->fs_dsize;
        sbp->f_bfree = fs->fs_cstotal.cs_nbfree * fs->fs_frag +
            fs->fs_cstotal.cs_nffree;
        sbp->f_bavail = sbp->f_bfree -
            ((int64_t)fs->fs_dsize * fs->fs_minfree / 100);
        sbp->f_files = fs->fs_ncg * fs->fs_ipg - ROOTINO;
        sbp->f_ffree = fs->fs_cstotal.cs_nifree;
        sbp->f_favail = sbp->f_ffree;
        copy_statfs_info(sbp, mp);

        return (0);
}

struct ffs_sync_args {
        int allerror;
        struct proc *p;
        int waitfor;
        int nlink0;
        int inflight;
        struct ucred *cred;
};

int
ffs_sync_vnode(struct vnode *vp, void *arg)
{
        struct ffs_sync_args *fsa = arg;
        struct inode *ip;
        int error, nlink0 = 0;

        if (vp->v_type == VNON)
                return (0);

        ip = VTOI(vp);

        if (vp->v_inflight && !(vp->v_type == VCHR || vp->v_type == VBLK))
                fsa->inflight = MIN(fsa->inflight+1, 65536);

        /*
         * If unmounting or converting rw to ro, then stop deferring
         * timestamp writes.
         */
        if (fsa->waitfor == MNT_WAIT && (ip->i_flag & IN_LAZYMOD)) {
                ip->i_flag |= IN_MODIFIED;
                UFS_UPDATE(ip, 1);
        }

        if (ip->i_effnlink == 0)
                nlink0 = 1;

        if ((ip->i_flag &
            (IN_ACCESS | IN_CHANGE | IN_MODIFIED | IN_UPDATE)) == 0 &&
            LIST_EMPTY(&vp->v_dirtyblkhd)) {
                goto end;
        }

        if (vget(vp, LK_EXCLUSIVE | LK_NOWAIT)) {
                nlink0 = 1;             /* potentially.. */
                goto end;
        }

        if ((error = VOP_FSYNC(vp, fsa->cred, fsa->waitfor, fsa->p)))
                fsa->allerror = error;
        VOP_UNLOCK(vp);
        vrele(vp);

end:
        fsa->nlink0 = MIN(fsa->nlink0 + nlink0, 65536);
        return (0);
}

/*
 * Go through the disk queues to initiate sandbagged IO;
 * go through the inodes to write those that have been modified;
 * initiate the writing of the super block if it has been modified.
 *
 * Should always be called with the mount point locked.
 */
int
ffs_sync(struct mount *mp, int waitfor, int stall, struct ucred *cred, struct proc *p)
{
        struct ufsmount *ump = VFSTOUFS(mp);
        struct fs *fs;
        int error, allerror = 0, count, clean, fmod;
        struct ffs_sync_args fsa;

        fs = ump->um_fs;
        /*
         * Write back modified superblock.
         * Consistency check that the superblock
         * is still in the buffer cache.
         */
        if (fs->fs_fmod != 0 && fs->fs_ronly != 0) {
                printf("fs = %s\n", fs->fs_fsmnt);
                panic("update: rofs mod");
        }
 loop:
        /*
         * Write back each (modified) inode.
         */
        fsa.allerror = 0;
        fsa.p = p;
        fsa.cred = cred;
        fsa.waitfor = waitfor;
        fsa.nlink0 = 0;
        fsa.inflight = 0;

        /*
         * Don't traverse the vnode list if we want to skip all of them.
         */
        if (waitfor != MNT_LAZY) {
                vfs_mount_foreach_vnode(mp, ffs_sync_vnode, &fsa);
                allerror = fsa.allerror;
        }

        /*
         * Force stale file system control information to be flushed.
         */
        if ((ump->um_mountp->mnt_flag & MNT_SOFTDEP) && waitfor == MNT_WAIT) {
                if ((error = softdep_flushworklist(ump->um_mountp, &count, p)))
                        allerror = error;
                /* Flushed work items may create new vnodes to clean */
                if (count)
                        goto loop;
        }
        if (waitfor != MNT_LAZY) {
                vn_lock(ump->um_devvp, LK_EXCLUSIVE | LK_RETRY);
                if ((error = VOP_FSYNC(ump->um_devvp, cred, waitfor, p)) != 0)
                        allerror = error;
                VOP_UNLOCK(ump->um_devvp);
        }
        qsync(mp);
        /*
         * Write back modified superblock.
         */
        clean = fs->fs_clean;
        fmod = fs->fs_fmod;
        if (stall && fs->fs_ronly == 0) {
                fs->fs_fmod = 1;
                if (allerror == 0 && fsa.nlink0 == 0 && fsa.inflight == 0) {
                        fs->fs_clean = (fs->fs_flags & FS_UNCLEAN) ? 0 : 1;
```

```c
#if 0
                        printf("%s force clean (dangling %d inflight %d)\n",
                            mp->mnt_stat.f_mntonname, fsa.nlink0, fsa.inflight);
#endif
                } else {
                        fs->fs_clean = 0;
#if 0
                        printf("%s force dirty (dangling %d inflight %d)\n",
                            mp->mnt_stat.f_mntonname, fsa.nlink0, fsa.inflight);
#endif
                }
        }
        if (fs->fs_fmod != 0 && (error = ffs_sbupdate(ump, waitfor)) != 0)
                allerror = error;
        fs->fs_clean = clean;
        fs->fs_fmod = fmod;

        return (allerror);
}

/*
 * Look up a FFS dinode number to find its incore vnode, otherwise read it
 * in from disk.  If it is in core, wait for the lock bit to clear, then
 * return the inode locked.  Detection and handling of mount points must be
 * done by the calling routine.
 */
int
ffs_vget(struct mount *mp, ino_t ino, struct vnode **vpp)
{
        struct fs *fs;
        struct inode *ip;
        struct ufs1_dinode *dp1;
#ifdef FFS2
        struct ufs2_dinode *dp2;
#endif
        struct ufsmount *ump;
        struct buf *bp;
        struct vnode *vp;
        dev_t dev;
        int error;

        if (ino > (ufsino_t)-1)
                panic("ffs_vget: alien ino_t %llu", (unsigned long long)ino);

        ump = VFSTOUFS(mp);
        dev = ump->um_dev;
retry:
        if ((*vpp = ufs_ihashget(dev, ino)) != NULL)
                return (0);

        /* Allocate a new vnode/inode. */
        if ((error = getnewvnode(VT_UFS, mp, &ffs_vops, &vp)) != 0) {
                *vpp = NULL;
                return (error);
        }

#ifdef VFSLCKDEBUG
        vp->v_flag |= VLOCKSWORK;
#endif
        ip = pool_get(&ffs_ino_pool, PR_WAITOK|PR_ZERO);
        rrw_init_flags(&ip->i_lock, "inode", RWL_DUPOK | RWL_IS_VNODE);
        ip->i_ump = ump;
        vref(ip->i_devvp);
        vp->v_data = ip;
        ip->i_vnode = vp;
        ip->i_fs = fs = ump->um_fs;
        ip->i_dev = dev;
        ip->i_number = ino;
        ip->i_vtbl = &ffs_vtbl;

        /*
         * Put it onto its hash chain and lock it so that other requests for
         * this inode will block if they arrive while we are sleeping waiting
         * for old data structures to be purged or for the contents of the
         * disk portion of this inode to be read.
         */
        error = ufs_ihashins(ip);

        if (error) {
                /*
                 * VOP_INACTIVE will treat this as a stale file
                 * and recycle it quickly
                 */
                vrele(vp);

                if (error == EEXIST)
                        goto retry;

                return (error);
        }


        /* Read in the disk contents for the inode, copy into the inode. */
        error = bread(ump->um_devvp, fsbtodb(fs, ino_to_fsba(fs, ino)),
            (int)fs->fs_bsize, &bp);
        if (error) {
                /*
                 * The inode does not contain anything useful, so it would
                 * be misleading to leave it on its hash chain. With mode
                 * still zero, it will be unlinked and returned to the free
                 * list by vput().
                 */
                vput(vp);
                brelse(bp);
                *vpp = NULL;
                return (error);
        }

#ifdef FFS2
        if (ip->i_ump->um_fstype == UM_UFS2) {
                ip->i_din2 = pool_get(&ffs_dinode2_pool, PR_WAITOK);
                dp2 = (struct ufs2_dinode *) bp->b_data + ino_to_fsbo(fs, ino);
                *ip->i_din2 = *dp2;
        } else
#endif
        {
                ip->i_din1 = pool_get(&ffs_dinode1_pool, PR_WAITOK);
                dp1 = (struct ufs1_dinode *) bp->b_data + ino_to_fsbo(fs, ino);
                *ip->i_din1 = *dp1;
        }

        brelse(bp);

        if (DOINGSOFTDEP(vp))
                softdep_load_inodeblock(ip);
        else
                ip->i_effnlink = DIP(ip, nlink);

        /*
         * Initialize the vnode from the inode, check for aliases.
         * Note that the underlying vnode may have changed.
         */
        if ((error = ffs_vinit(mp, &vp)) != 0) {
                vput(vp);
                *vpp = NULL;
                return (error);
        }

        /*
         * Set up a generation number for this inode if it does not
         * already have one. This should only happen on old filesystems.
         */
        if (DIP(ip, gen) == 0) {
                while (DIP(ip, gen) == 0)
                        DIP_ASSIGN(ip, gen, arc4random());
                if ((vp->v_mount->mnt_flag & MNT_RDONLY) == 0)
                        ip->i_flag |= IN_MODIFIED;
        }

        /*
         * Ensure that uid and gid are correct. This is a temporary
         * fix until fsck has been changed to do the update.
         */
        if (fs->fs_magic == FS_UFS1_MAGIC && fs->fs_inodefmt < FS_44INODEFMT) {
                ip->i_ffs1_uid = ip->i_din1->di_ouid;
                ip->i_ffs1_gid = ip->i_din1->di_ogid;
        }

        *vpp = vp;

        return (0);
}

/*
 * File handle to vnode
 *
 * Have to be really careful about stale file handles.
 */
int
ffs_fhtovp(struct mount *mp, struct fid *fhp, struct vnode **vpp)
{
        struct ufid *ufhp;
        int error;

        ufhp = (struct ufid *)fhp;
        if (ufhp->ufid_len != sizeof(*ufhp))
                return EINVAL;

        if ((error = ffs_checkrange(mp, ufhp->ufid_ino)) != 0)
                return error;

        return (ufs_fhtovp(mp, ufhp, vpp));
}

/*
 * Vnode pointer to File handle
 */
int
ffs_vptofh(struct vnode *vp, struct fid *fhp)
{
        struct inode *ip;
        struct ufid *ufhp;

        ip = VTOI(vp);
        ufhp = (struct ufid *)fhp;
        ufhp->ufid_len = sizeof(struct ufid);
        ufhp->ufid_ino = ip->i_number;
        ufhp->ufid_gen = DIP(ip, gen);

        return (0);
}

/*
 * Write a superblock and associated information back to disk.
 */
int
ffs_sbupdate(struct ufsmount *mp, int waitfor)
{
        struct fs *dfs, *fs = mp->um_fs;
        struct buf *bp;
        int blks;
        caddr_t space;
        int i, size, error, allerror = 0;

        /*
         * First write back the summary information.
         */
        blks = howmany(fs->fs_cssize, fs->fs_fsize);
        space = (caddr_t)fs->fs_csp;
        for (i = 0; i < blks; i += fs->fs_frag) {
                size = fs->fs_bsize;
                if (i + fs->fs_frag > blks)
                        size = (blks - i) * fs->fs_fsize;
                bp = getblk(mp->um_devvp, fsbtodb(fs, fs->fs_csaddr + i),
                    size, 0, INFSLP);
                memcpy(bp->b_data, space, size);
                space += size;
                if (waitfor != MNT_WAIT)
                        bawrite(bp);
                else if ((error = bwrite(bp)))
                        allerror = error;
        }

        /*
         * Now write back the superblock itself. If any errors occurred
         * up to this point, then fail so that the superblock avoids
         * being written out as clean.
         */
        if (allerror) {
                return (allerror);
        }

        bp = getblk(mp->um_devvp,
            fs->fs_sblockloc >> (fs->fs_fshift - fs->fs_fsbtodb),
            (int)fs->fs_sbsize, 0, INFSLP);
        fs->fs_fmod = 0;
        fs->fs_time = gettime();
        memcpy(bp->b_data, fs, fs->fs_sbsize);
        /* Restore compatibility to old file systems.              XXX */
        dfs = (struct fs *)bp->b_data;                          /* XXX */
        if (fs->fs_postblformat == FS_42POSTBLFMT)      /* XXX */
                dfs->fs_nrpos = -1;                             /* XXX */
        if (fs->fs_inodefmt < FS_44INODEFMT) {                  /* XXX */
                int32_t *lp, tmp;                               /* XXX */
                                                                /* XXX */
                lp = (int32_t *)&dfs->fs_qbmask;        /* XXX */
                tmp = lp[4];                                    /* XXX */
                for (i = 4; i > 0; i--)                         /* XXX */
                        lp[i] = lp[i-1];                /* XXX */
                lp[0] = tmp;                                    /* XXX */
        }                                                       /* XXX */
        dfs->fs_maxfilesize = mp->um_savedmaxfilesize;          /* XXX */

        ffs1_compat_write(dfs, mp);

        if (waitfor != MNT_WAIT)
                bawrite(bp);
        else if ((error = bwrite(bp)))
                allerror = error;

        return (allerror);
}

int
ffs_init(struct vfsconf *vfsp)
{
        static int done;

        if (done)
                return (0);

        done = 1;

        pool_init(&ffs_ino_pool, sizeof(struct inode), 0, IPL_NONE,
            PR_WAITOK, "ffsino", NULL);
        pool_init(&ffs_dinode1_pool, sizeof(struct ufs1_dinode), 0, IPL_NONE,
            PR_WAITOK, "dino1pl", NULL);
#ifdef FFS2
        pool_init(&ffs_dinode2_pool, sizeof(struct ufs2_dinode), 0, IPL_NONE,
            PR_WAITOK, "dino2pl", NULL);
#endif

        softdep_initialize();

        return (ufs_init(vfsp));
}

#ifdef FFS_SOFTUPDATES
extern int max_softdeps, tickdelay, stat_worklist_push;
extern int stat_blk_limit_push, stat_ino_limit_push, stat_blk_limit_hit;
extern int stat_ino_limit_hit, stat_sync_limit_hit, stat_indir_blk_ptrs;
extern int stat_inode_bitmap, stat_direct_blk_ptrs, stat_dir_entry;
#endif
const struct sysctl_bounded_args ffs_vars[] = {
```

```c
#ifdef FFS_SOFTUPDATES
        { FFS_MAX_SOFTDEPS, &max_softdeps, 0, INT_MAX },
        { FFS_SD_TICKDELAY, &tickdelay, 2, INT_MAX },
        { FFS_SD_WORKLIST_PUSH, &stat_worklist_push, 1, 0 }, /* read-only */
        { FFS_SD_BLK_LIMIT_PUSH, &stat_blk_limit_push, 1, 0 },
        { FFS_SD_INO_LIMIT_PUSH, &stat_ino_limit_push, 1, 0 },
        { FFS_SD_BLK_LIMIT_HIT, &stat_blk_limit_hit, 1, 0 },
        { FFS_SD_INO_LIMIT_HIT, &stat_ino_limit_hit, 1, 0 },
        { FFS_SD_SYNC_LIMIT_HIT, &stat_sync_limit_hit, 1, 0 },
        { FFS_SD_INDIR_BLK_PTRS, &stat_indir_blk_ptrs, 1, 0 },
        { FFS_SD_INODE_BITMAP, &stat_inode_bitmap, 1, 0 },
        { FFS_SD_DIRECT_BLK_PTRS, &stat_direct_blk_ptrs, 1, 0 },
        { FFS_SD_DIR_ENTRY, &stat_dir_entry, 1, 0 },
#endif
#ifdef UFS_DIRHASH
        { FFS_DIRHASH_DIRSIZE, &ufs_mindirhashsize, 0, INT_MAX },
        { FFS_DIRHASH_MAXMEM, &ufs_dirhashmaxmem, 0, INT_MAX },
        { FFS_DIRHASH_MEM, &ufs_dirhashmem, 1, 0 },
#endif
};

/*
 * fast filesystem related variables.
 */
int
ffs_sysctl(int *name, u_int namelen, void *oldp, size_t *oldlenp, void *newp,
    size_t newlen, struct proc *p)
{
        return sysctl_bounded_arr(ffs_vars, nitems(ffs_vars), name,
            namelen, oldp, oldlenp, newp, newlen);
}
File: ffs_vnops.c


/*      $OpenBSD: ffs_vnops.c,v 1.98 2020/02/27 09:10:31 mpi Exp $ */
/*      $NetBSD: ffs_vnops.c,v 1.7 1996/05/11 18:27:24 mycroft Exp $        */

/*
 * Copyright (c) 1982, 1986, 1989, 1993
 *      The Regents of the University of California.  All rights reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions
 * are met:
 * 1. Redistributions of source code must retain the above copyright
 *    notice, this list of conditions and the following disclaimer.
 * 2. Redistributions in binary form must reproduce the above copyright
 *    notice, this list of conditions and the following disclaimer in the
 *    documentation and/or other materials provided with the distribution.
 * 3. Neither the name of the University nor the names of its contributors
 *    may be used to endorse or promote products derived from this software
 *    without specific prior written permission.
 *
 * THIS SOFTWARE IS PROVIDED BY THE REGENTS AND CONTRIBUTORS ``AS IS'' AND
 * ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
 * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
 * ARE DISCLAIMED.  IN NO EVENT SHALL THE REGENTS OR CONTRIBUTORS BE LIABLE
 * FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
 * DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
 * OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
 * HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
 * LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
 * OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
 * SUCH DAMAGE.
 *
 *      @(#)ffs_vnops.c  8.10 (Berkeley) 8/10/94
 */

#include <sys/param.h>
#include <sys/systm.h>
#include <sys/resourcevar.h>
#include <sys/kernel.h>
#include <sys/stat.h>
#include <sys/buf.h>
#include <sys/mount.h>
#include <sys/vnode.h>
#include <sys/malloc.h>
#include <sys/signalvar.h>
#include <sys/pool.h>
#include <sys/event.h>
#include <sys/specdev.h>

#include <miscfs/fifofs/fifo.h>

#include <ufs/ufs/quota.h>
#include <ufs/ufs/inode.h>
#include <ufs/ufs/dir.h>
#include <ufs/ufs/ufs_extern.h>
#include <ufs/ufs/ufsmount.h>

#include <ufs/ffs/fs.h>
#include <ufs/ffs/ffs_extern.h>

const struct vops ffs_vops = {
        .vop_lookup     = ufs_lookup,
        .vop_create     = ufs_create,
        .vop_mknod      = ufs_mknod,
        .vop_open       = ufs_open,
        .vop_close      = ufs_close,
        .vop_access     = ufs_access,
        .vop_getattr    = ufs_getattr,
        .vop_setattr    = ufs_setattr,
        .vop_read       = ffs_read,
        .vop_write      = ffs_write,
        .vop_ioctl      = ufs_ioctl,
        .vop_poll       = ufs_poll,
        .vop_kqfilter   = ufs_kqfilter,
        .vop_revoke     = vop_generic_revoke,
        .vop_fsync      = ffs_fsync,
        .vop_remove     = ufs_remove,
        .vop_link       = ufs_link,
        .vop_rename     = ufs_rename,
        .vop_mkdir      = ufs_mkdir,
        .vop_rmdir      = ufs_rmdir,
        .vop_symlink    = ufs_symlink,
        .vop_readdir    = ufs_readdir,
        .vop_readlink   = ufs_readlink,
        .vop_abortop    = vop_generic_abortop,
        .vop_inactive   = ufs_inactive,
        .vop_reclaim    = ffs_reclaim,
        .vop_lock       = ufs_lock,
        .vop_unlock     = ufs_unlock,
        .vop_bmap       = ufs_bmap,
        .vop_strategy   = ufs_strategy,
        .vop_print      = ufs_print,
        .vop_islocked   = ufs_islocked,
        .vop_pathconf   = ufs_pathconf,
        .vop_advlock    = ufs_advlock,
        .vop_bwrite     = vop_generic_bwrite
};

const struct vops ffs_specvops = {
        .vop_close      = ufsspec_close,
        .vop_access     = ufs_access,
        .vop_getattr    = ufs_getattr,
        .vop_setattr    = ufs_setattr,
        .vop_read       = ufsspec_read,
        .vop_write      = ufsspec_write,
        .vop_fsync      = ffs_fsync,
        .vop_inactive   = ufs_inactive,
        .vop_reclaim    = ffs_reclaim,
        .vop_lock       = ufs_lock,
        .vop_unlock     = ufs_unlock,
        .vop_print      = ufs_print,
        .vop_islocked   = ufs_islocked,

        /* XXX: Keep in sync with spec_vops */
        .vop_lookup     = vop_generic_lookup,
        .vop_create     = spec_badop,
        .vop_mknod      = spec_badop,
        .vop_open       = spec_open,
        .vop_ioctl      = spec_ioctl,
        .vop_poll       = spec_poll,
        .vop_kqfilter   = spec_kqfilter,
        .vop_revoke     = vop_generic_revoke,
        .vop_remove     = spec_badop,
        .vop_link       = spec_badop,
        .vop_rename     = spec_badop,
        .vop_mkdir      = spec_badop,
        .vop_rmdir      = spec_badop,
        .vop_symlink    = spec_badop,
        .vop_readdir    = spec_badop,
        .vop_readlink   = spec_badop,
        .vop_abortop    = spec_badop,
        .vop_bmap       = vop_generic_bmap,
        .vop_strategy   = spec_strategy,
        .vop_pathconf   = spec_pathconf,
        .vop_advlock    = spec_advlock,
        .vop_bwrite     = vop_generic_bwrite,
};

#ifdef FIFO
const struct vops ffs_fifovops = {
        .vop_close      = ufsfifo_close,
        .vop_access     = ufs_access,
        .vop_getattr    = ufs_getattr,
        .vop_setattr    = ufs_setattr,
        .vop_read       = ufsfifo_read,
        .vop_write      = ufsfifo_write,
        .vop_fsync      = ffs_fsync,
        .vop_inactive   = ufs_inactive,
        .vop_reclaim    = ffsfifo_reclaim,
        .vop_lock       = ufs_lock,
        .vop_unlock     = ufs_unlock,
        .vop_print      = ufs_print,
        .vop_islocked   = ufs_islocked,
        .vop_bwrite     = vop_generic_bwrite,

        /* XXX: Keep in sync with fifo_vops */
        .vop_lookup     = vop_generic_lookup,
        .vop_create     = fifo_badop,
        .vop_mknod      = fifo_badop,
        .vop_open       = fifo_open,
        .vop_ioctl      = fifo_ioctl,
        .vop_poll       = fifo_poll,
        .vop_kqfilter   = fifo_kqfilter,
        .vop_revoke     = vop_generic_revoke,
        .vop_remove     = fifo_badop,
        .vop_link       = fifo_badop,
        .vop_rename     = fifo_badop,
        .vop_mkdir      = fifo_badop,
        .vop_rmdir      = fifo_badop,
        .vop_symlink    = fifo_badop,
        .vop_readdir    = fifo_badop,
        .vop_readlink   = fifo_badop,
        .vop_abortop    = fifo_badop,
        .vop_bmap       = vop_generic_bmap,
        .vop_strategy   = fifo_badop,
        .vop_pathconf   = fifo_pathconf,
        .vop_advlock    = fifo_advlock
};
#endif /* FIFO */

/*
 * Vnode op for reading.
 */
int
ffs_read(void *v)
{
        struct vop_read_args *ap = v;
        struct vnode *vp;
        struct inode *ip;
        struct uio *uio;
        struct fs *fs;
        struct buf *bp;
        daddr_t lbn, nextlbn;
        off_t bytesinfile;
        int size, xfersize, blkoffset;
        mode_t mode;
        int error;

        vp = ap->a_vp;
        ip = VTOI(vp);
        mode = DIP(ip, mode);
        uio = ap->a_uio;

#ifdef DIAGNOSTIC
        if (uio->uio_rw != UIO_READ)
                panic("ffs_read: mode");

        if (vp->v_type == VLNK) {
                if (DIP(ip, size) < ip->i_ump->um_maxsymlinklen ||
                    (ip->i_ump->um_maxsymlinklen == 0 && DIP(ip, blocks) == 0))
                        panic("ffs_read: short symlink");
        } else if (vp->v_type != VREG && vp->v_type != VDIR)
                panic("ffs_read: type %d", vp->v_type);
#endif
        fs = ip->i_fs;
        if (uio->uio_offset < 0)
                return (EINVAL);
        if (uio->uio_resid == 0)
                return (0);

        for (error = 0, bp = NULL; uio->uio_resid > 0; bp = NULL) {
                if ((bytesinfile = DIP(ip, size) - uio->uio_offset) <= 0)
                        break;
                lbn = lblkno(fs, uio->uio_offset);
                nextlbn = lbn + 1;
                size = fs->fs_bsize;    /* WAS blksize(fs, ip, lbn); */
                blkoffset = blkoff(fs, uio->uio_offset);
                xfersize = fs->fs_bsize - blkoffset;
                if (uio->uio_resid < xfersize)
                        xfersize = uio->uio_resid;
                if (bytesinfile < xfersize)
                        xfersize = bytesinfile;

                if (lblktosize(fs, nextlbn) >= DIP(ip, size))
                        error = bread(vp, lbn, size, &bp);
                else if (lbn - 1 == ip->i_ci.ci_lastr ||
                    uio->uio_resid > xfersize) {
                        error = bread_cluster(vp, lbn, size, &bp);
                } else
                        error = bread(vp, lbn, size, &bp);

                if (error)
                        break;
                ip->i_ci.ci_lastr = lbn;

                /*
                 * We should only get non-zero b_resid when an I/O error
                 * has occurred, which should cause us to break above.
                 * However, if the short read did not cause an error,
                 * then we want to ensure that we do not uiomove bad
                 * or uninitialized data.
                 */
                size -= bp->b_resid;
                if (size < xfersize) {
                        if (size == 0)
                                break;
                        xfersize = size;
                }
                error = uiomove(bp->b_data + blkoffset, xfersize, uio);
                if (error)
                        break;
                brelse(bp);
        }
        if (bp != NULL)
                brelse(bp);
        if (!(vp->v_mount->mnt_flag & MNT_NOATIME) ||
            (ip->i_flag & (IN_CHANGE | IN_UPDATE))) {
                ip->i_flag |= IN_ACCESS;
        }
        return (error);
}
```

```c
/*
 * Vnode op for writing.
 */
int
ffs_write(void *v)
{
        struct vop_write_args *ap = v;
        struct vnode *vp;
        struct uio *uio;
        struct inode *ip;
        struct fs *fs;
        struct buf *bp;
        daddr_t lbn;
        off_t osize;
        int blkoffset, error, extended, flags, ioflag, size, xfersize;
        size_t resid;
        ssize_t overrun;

        extended = 0;
        ioflag = ap->a_ioflag;
        uio = ap->a_uio;
        vp = ap->a_vp;
        ip = VTOI(vp);

#ifdef DIAGNOSTIC
        if (uio->uio_rw != UIO_WRITE)
                panic("ffs_write: mode");
#endif

        /*
         * If writing 0 bytes, succeed and do not change
         * update time or file offset (standards compliance)
         */
        if (uio->uio_resid == 0)
                return (0);

        switch (vp->v_type) {
        case VREG:
                if (ioflag & IO_APPEND)
                        uio->uio_offset = DIP(ip, size);
                if ((DIP(ip, flags) & APPEND) && uio->uio_offset != DIP(ip, size))
                        return (EPERM);
                /* FALLTHROUGH */
        case VLNK:
                break;
        case VDIR:
                if ((ioflag & IO_SYNC) == 0)
                        panic("ffs_write: nonsync dir write");
                break;
        default:
                panic("ffs_write: type %d", vp->v_type);
        }

        fs = ip->i_fs;
        if (uio->uio_offset < 0 ||
            (u_int64_t)uio->uio_offset + uio->uio_resid > fs->fs_maxfilesize)
                return (EFBIG);

        /* do the filesize rlimit check */
        if ((error = vn_fsizechk(vp, uio, ioflag, &overrun)))
                return (error);

        resid = uio->uio_resid;
        osize = DIP(ip, size);
        flags = ioflag & IO_SYNC ? B_SYNC : 0;

        for (error = 0; uio->uio_resid > 0;) {
                lbn = lblkno(fs, uio->uio_offset);
                blkoffset = blkoff(fs, uio->uio_offset);
                xfersize = fs->fs_bsize - blkoffset;
                if (uio->uio_resid < xfersize)
                        xfersize = uio->uio_resid;
                if (fs->fs_bsize > xfersize)
                        flags |= B_CLRBUF;
                else
                        flags &= ~B_CLRBUF;

                if ((error = UFS_BUF_ALLOC(ip, uio->uio_offset, xfersize,
                         ap->a_cred, flags, &bp)) != 0)
                        break;
                if (uio->uio_offset + xfersize > DIP(ip, size)) {
                        DIP_ASSIGN(ip, size, uio->uio_offset + xfersize);
                        uvm_vnp_setsize(vp, DIP(ip, size));
                        extended = 1;
                }
                (void)uvm_vnp_uncache(vp);

                size = blksize(fs, ip, lbn) - bp->b_resid;
                if (size < xfersize)
                        xfersize = size;

                error = uiomove(bp->b_data + blkoffset, xfersize, uio);
                /*
                 * If the buffer is not already filled and we encounter an
                 * error while trying to fill it, we have to clear out any
                 * garbage data from the pages instantiated for the buffer.
                 * If we do not, a failed uiomove() during a write can leave
                 * the prior contents of the pages exposed to a userland mmap.
                 *
                 * Note that we don't need to clear buffers that were
                 * allocated with the B_CLRBUF flag set.
                */
                if (error != 0 && !(flags & B_CLRBUF))
                        memset(bp->b_data + blkoffset, 0, xfersize);

                if (ioflag & IO_NOCACHE)
                        bp->b_flags |= B_NOCACHE;

                if (ioflag & IO_SYNC)
                        (void)bwrite(bp);
                else if (xfersize + blkoffset == fs->fs_bsize) {
                        bawrite(bp);
                } else
                        bdwrite(bp);

                if (error || xfersize == 0)
                        break;
                ip->i_flag |= IN_CHANGE | IN_UPDATE;
        }
        /*
         * If we successfully wrote any data, and we are not the superuser
         * we clear the setuid and setgid bits as a precaution against
         * tampering.
         */
        if (resid > uio->uio_resid && ap->a_cred && ap->a_cred->cr_uid != 0 &&
            !vnoperm(vp))
                DIP_ASSIGN(ip, mode, DIP(ip, mode) & ~(ISUID | ISGID));
        if (resid > uio->uio_resid)
                VN_KNOTE(vp, NOTE_WRITE | (extended ? NOTE_EXTEND : 0));
        if (error) {
                if (ioflag & IO_UNIT) {
                        (void)UFS_TRUNCATE(ip, osize,
                            ioflag & IO_SYNC, ap->a_cred);
                        uio->uio_offset -= resid - uio->uio_resid;
                        uio->uio_resid = resid;
                }
        } else if (resid > uio->uio_resid && (ioflag & IO_SYNC)) {
                error = UFS_UPDATE(ip, 1);
        }
        /* correct the result for writes clamped by vn_fsizechk() */
        uio->uio_resid += overrun;
        return (error);
}

/*
 * Synch an open file.
 */
int
ffs_fsync(void *v)
{
        struct vop_fsync_args *ap = v;
        struct vnode *vp = ap->a_vp;
        struct buf *bp, *nbp;
        int s, error, passes, skipmeta;

        if (vp->v_type == VBLK &&
            vp->v_specmountpoint != NULL &&
            (vp->v_specmountpoint->mnt_flag & MNT_SOFTDEP))
                softdep_fsync_mountdev(vp, ap->a_waitfor);

        /*
         * Flush all dirty buffers associated with a vnode.
         */
        passes = NIADDR + 1;
        skipmeta = 0;
        if (ap->a_waitfor == MNT_WAIT)
                skipmeta = 1;
        s = splbio();
loop:
        LIST_FOREACH(bp, &vp->v_dirtyblkhd, b_vnbufs) {
                bp->b_flags &= ~B_SCANNED;
        }
        LIST_FOREACH_SAFE(bp, &vp->v_dirtyblkhd, b_vnbufs, nbp) {
                /*
                 * Reasons to skip this buffer: it has already been considered
                 * on this pass, this pass is the first time through on a
                 * synchronous flush request and the buffer being considered
                 * is metadata, the buffer has dependencies that will cause
                 * it to be redirtied and it has not already been deferred,
                 * or it is already being written.
                 */
                if (bp->b_flags & (B_BUSY | B_SCANNED))
                        continue;
                if ((bp->b_flags & B_DELWRI) == 0)
                        panic("ffs_fsync: not dirty");
                if (skipmeta && bp->b_lblkno < 0)
                        continue;
                if (ap->a_waitfor != MNT_WAIT &&
                    LIST_FIRST(&bp->b_dep) != NULL &&
                    (bp->b_flags & B_DEFERRED) == 0 &&
                    buf_countdeps(bp, 0, 1)) {
                        bp->b_flags |= B_DEFERRED;
                        continue;
                }

                bremfree(bp);
                buf_acquire(bp);
                bp->b_flags |= B_SCANNED;
                splx(s);
                /*
                 * On our final pass through, do all I/O synchronously
                 * so that we can find out if our flush is failing
                 * because of write errors.
                 */
                if (passes > 0 || ap->a_waitfor != MNT_WAIT)
                        (void) bawrite(bp);
                else if ((error = bwrite(bp)) != 0)
                        return (error);
                s = splbio();
                /*
                 * Since we may have slept during the I/O, we need
                 * to start from a known point.
                 */
                nbp = LIST_FIRST(&vp->v_dirtyblkhd);
        }
        if (skipmeta) {
                skipmeta = 0;
                goto loop;
        }
        if (ap->a_waitfor == MNT_WAIT) {
                vwaitforio(vp, 0, "ffs_fsync", INFSLP);

                /*
                 * Ensure that any filesystem metadata associated
                 * with the vnode has been written.
                 */
                splx(s);
                if ((error = softdep_sync_metadata(ap)) != 0)
                        return (error);
                s = splbio();
                if (!LIST_EMPTY(&vp->v_dirtyblkhd)) {
                        /*
                         * Block devices associated with filesystems may
                         * have new I/O requests posted for them even if
                         * the vnode is locked, so no amount of trying will
                         * get them clean. Thus we give block devices a
                         * good effort, then just give up. For all other file
                         * types, go around and try again until it is clean.
                         */
                        if (passes > 0) {
                                passes -= 1;
                                goto loop;
                        }
#ifdef DIAGNOSTIC
                        if (vp->v_type != VBLK)
                                vprint("ffs_fsync: dirty", vp);
#endif
                }
        }
        splx(s);
        return (UFS_UPDATE(VTOI(vp), ap->a_waitfor == MNT_WAIT));
}

/*
 * Reclaim an inode so that it can be used for other purposes.
 */
int
ffs_reclaim(void *v)
{
        struct vop_reclaim_args *ap = v;
        struct vnode *vp = ap->a_vp;
        struct inode *ip = VTOI(vp);
        int error;

        if ((error = ufs_reclaim(vp)) != 0)
                return (error);

        if (ip->i_din1 != NULL) {
#ifdef FFS2
                if (ip->i_ump->um_fstype == UM_UFS2)
                        pool_put(&ffs_dinode2_pool, ip->i_din2);
                else
#endif
                        pool_put(&ffs_dinode1_pool, ip->i_din1);
        }

        pool_put(&ffs_ino_pool, ip);

        vp->v_data = NULL;

        return (0);
}

#ifdef FIFO
int
ffsfifo_reclaim(void *v)
{
        fifo_reclaim(v);
        return (ffs_reclaim(v));
}
#endif
```

File: fs.h

```c
/*      $OpenBSD: fs.h,v 1.43 2020/06/20 07:49:04 otto Exp $       */
/*      $NetBSD: fs.h,v 1.6 1995/04/12 21:21:02 mycroft Exp $      */

/*
 * Copyright (c) 1982, 1986, 1993
 *      The Regents of the University of California.  All rights reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions
 * are met:
 * 1. Redistributions of source code must retain the above copyright
 *    notice, this list of conditions and the following disclaimer.
 * 2. Redistributions in binary form must reproduce the above copyright
 *    notice, this list of conditions and the following disclaimer in the
 *    documentation and/or other materials provided with the distribution.
 * 3. Neither the name of the University nor the names of its contributors
 *    may be used to endorse or promote products derived from this software
 *    without specific prior written permission.
 *
 * THIS SOFTWARE IS PROVIDED BY THE REGENTS AND CONTRIBUTORS ``AS IS'' AND
 * ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
 * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
 * ARE DISCLAIMED.  IN NO EVENT SHALL THE REGENTS OR CONTRIBUTORS BE LIABLE
 * FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
 * DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
 * OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
 * HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
 * LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
 * OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
 * SUCH DAMAGE.
 *
 *      @(#)fs.h 8.10 (Berkeley) 10/27/94
 */

/*
 * Each disk drive contains some number of file systems.
 * A file system consists of a number of cylinder groups.
 * Each cylinder group has inodes and data.
 *
 * A file system is described by its super-block, which in turn
 * describes the cylinder groups.  The super-block is critical
 * data and is replicated in each cylinder group to protect against
 * catastrophic loss.  This is done at `newfs' time and the critical
 * super-block data does not change, so the copies need not be
 * referenced further unless disaster strikes.
 *
 * For file system fs, the offsets of the various blocks of interest
 * are given in the super block as:
 *      [fs->fs_sblkno]         Super-block
 *      [fs->fs_cblkno]         Cylinder group block
 *      [fs->fs_iblkno]         Inode blocks
 *      [fs->fs_dblkno]         Data blocks
 * The beginning of cylinder group cg in fs, is given by
 * the ``cgbase(fs, cg)'' macro.
 *
 * The first boot and super blocks are given in absolute disk addresses.
 * The byte-offset forms are preferred, as they don't imply a sector size.
 */
#define BBSIZE          8192
#define SBSIZE          8192
#define  BBOFF          ((off_t)(0))
#define  SBOFF          ((off_t)(BBOFF + BBSIZE))
#define  BBLOCK         ((daddr_t)(0))
#define  SBLOCK         ((daddr_t)(BBLOCK + BBSIZE / DEV_BSIZE))
#define  SBLOCK_UFS1    8192
#define  SBLOCK_UFS2    65536
#define  SBLOCK_PIGGY   262144
#define  SBLOCKSIZE     8192
#define  SBLOCKSEARCH \
        { SBLOCK_UFS2, SBLOCK_UFS1, SBLOCK_PIGGY, -1 }

/*
 * Addresses stored in inodes are capable of addressing fragments
 * of `blocks'. File system blocks of at most size MAXBSIZE can
 * be optionally broken into 2, 4, or 8 pieces, each of which is
 * addressible; these pieces may be DEV_BSIZE, or some multiple of
 * a DEV_BSIZE unit.
 *
 * Large files consist of exclusively large data blocks.  To avoid
 * undue wasted disk space, the last data block of a small file may be
 * allocated as only as many fragments of a large block as are
 * necessary.  The file system format retains only a single pointer
 * to such a fragment, which is a piece of a single large block that
 * has been divided.  The size of such a fragment is determinable from
 * information in the inode, using the ``blksize(fs, ip, lbn)'' macro.
 *
 * The file system records space availability at the fragment level;
 * to determine block availability, aligned fragments are examined.
 */

#define MAXFRAG  8

/*
 * MINBSIZE is the smallest allowable block size.
 * In order to insure that it is possible to create files of size
 * 2^32 with only two levels of indirection, MINBSIZE is set to 4096.
 * MINBSIZE must be big enough to hold a cylinder group block,
 * thus changes to (struct cg) must keep its size within MINBSIZE.
 * Note that super blocks are always of size SBSIZE,
 * and that both SBSIZE and MAXBSIZE must be >= MINBSIZE.
 */
#define MINBSIZE 4096

/*
 * The path name on which the file system is mounted is maintained
 * in fs_fsmnt. MAXMNTLEN defines the amount of space allocated in
 * the super block for this name.
 */
#define MAXMNTLEN       468

/*
 * The volume name for this file system is kept in fs_volname.
 * MAXVOLLEN defines the length of the buffer allocated.
 */
#define MAXVOLLEN       32

/*
 * There is a 128-byte region in the superblock reserved for in-core
 * pointers to summary information. Originally this included an array
 * of pointers to blocks of struct csum; now there are just three
 * pointers and the remaining space is padded with fs_ocsp[].
 *
 * NOCSPTRS determines the size of this padding. One pointer (fs_csp)
 * is taken away to point to a contiguous array of struct csum for
 * all cylinder groups; a second (fs_maxcluster) points to an array
 * of cluster sizes that is computed as cylinder groups are inspected,
 * and the third points to an array that tracks the creation of new
 * directories.
 */
#define NOCSPTRS ((128 / sizeof(void *)) - 4)

/*
 * A summary of contiguous blocks of various sizes is maintained
 * in each cylinder group. Normally this is set by the initial
 * value of fs_maxcontig. To conserve space, a maximum summary size
 * is set by FS_MAXCONTIG.
 */
#define FS_MAXCONTIG    16

/*
 * MINFREE gives the minimum acceptable percentage of file system
 * blocks which may be free. If the freelist drops below this level
 * only the superuser may continue to allocate blocks. This may
 * be set to 0 if no reserve of free blocks is deemed necessary,
 * however throughput drops by fifty percent if the file system
 * is run at between 95% and 100% full; thus the minimum default
 * value of fs_minfree is 5%. However, to get good clustering
 * performance, 10% is a better choice. With 5% free space,
 * fragmentation is not a problem, so we choose to optimize for time.
 */
#define MINFREE         5
#define DEFAULTOPT      FS_OPTTIME

/*
 * The directory preference algorithm(dirpref) can be tuned by adjusting
 * the following parameters which tell the system the average file size
 * and the average number of files per directory. These defaults are well
 * selected for typical filesystems, but may need to be tuned for odd
 * cases like filesystems being used for squid caches or news spools.
 */
#define AVFILESIZ       16384   /* expected average file size */
#define AFPDIR          64      /* expected number of files per directory */

/*
 * Size of superblock space reserved for snapshots.
 */
#define FSMAXSNAP       20

/*
 * Per cylinder group information; summarized in blocks allocated
 * from first cylinder group data blocks.  These blocks have to be
 * read in from fs_csaddr (size fs_cssize) in addition to the
 * super block.
 */
struct csum {
        int32_t  cs_ndir;       /* number of directories */
        int32_t  cs_nbfree;             /* number of free blocks */
        int32_t  cs_nifree;             /* number of free inodes */
        int32_t  cs_nffree;             /* number of free frags */
};

struct csum_total {
        int64_t cs_ndir;        /* number of directories */
        int64_t cs_nbfree;              /* number of free blocks */
        int64_t cs_nifree;              /* number of free inodes */
        int64_t cs_nffree;              /* number of free frags */
        int64_t cs_spare[4];            /* future expansion */
};

/*
 * Super block for an FFS file system.
 */
struct fs {
        int32_t  fs_firstfield;         /* historic file system linked list, */
        int32_t  fs_unused_1;           /*     used for incore super blocks */
        int32_t  fs_sblkno;             /* addr of super-block / frags */
        int32_t  fs_cblkno;             /* offset of cyl-block / frags */
        int32_t  fs_iblkno;             /* offset of inode-blocks / frags */
        int32_t  fs_dblkno;             /* offset of first data / frags */
        int32_t  fs_cgoffset;           /* cylinder group offset in cylinder */
        int32_t  fs_cgmask;             /* used to calc mod fs_ntrak */
        int32_t  fs_ffs1_time;          /* last time written */
        int32_t  fs_ffs1_size;          /* # of blocks in fs / frags */
        int32_t  fs_ffs1_dsize;         /* # of data blocks in fs */
        u_int32_t fs_ncg;               /* # of cylinder groups */
        int32_t  fs_bsize;              /* size of basic blocks / bytes */
        int32_t  fs_fsize;              /* size of frag blocks / bytes */
        int32_t  fs_frag;               /* # of frags in a block in fs */
/* these are configuration parameters */
        int32_t  fs_minfree;            /* minimum percentage of free blocks */
        int32_t  fs_rotdelay;           /* # of ms for optimal next block */
        int32_t  fs_rps;        /* disk revolutions per second */
/* these fields can be computed from the others */
        int32_t  fs_bmask;              /* ``blkoff'' calc of blk offsets */
        int32_t  fs_fmask;              /* ``fragoff'' calc of frag offsets */
        int32_t  fs_bshift;             /* ``lblkno'' calc of logical blkno */
        int32_t  fs_fshift;             /* ``numfrags'' calc # of frags */
/* these are configuration parameters */
        int32_t  fs_maxcontig;          /* max # of contiguous blks */
        int32_t  fs_maxbpg;             /* max # of blks per cyl group */
/* these fields can be computed from the others */
        int32_t  fs_fragshift;          /* block to frag shift */
        int32_t  fs_fsbtodb;            /* fsbtodb and dbtofsb shift constant */
        int32_t  fs_sbsize;             /* actual size of super block */
        int32_t  fs_csmask;             /* csum block offset (now unused) */
        int32_t  fs_csshift;            /* csum block number (now unused) */
        int32_t  fs_nindir;             /* value of NINDIR */
        u_int32_t fs_inopb;             /* inodes per file system block */
        int32_t  fs_nspf;               /* DEV_BSIZE sectors per frag */
/* yet another configuration parameter */
        int32_t  fs_optim;              /* optimization preference, see below */
/* these fields are derived from the hardware */
        int32_t  fs_npsect;             /* DEV_BSIZE sectors/track + spares */
        int32_t  fs_interleave;         /* DEV_BSIZE sector interleave */
        int32_t  fs_trackskew;          /* sector 0 skew, per track */
/* fs_id takes the space of the unused fs_headswitch and fs_trkseek fields */
        int32_t  fs_id[2];              /* unique filesystem id */
/* sizes determined by number of cylinder groups and their sizes */
        int32_t  fs_ffs1_csaddr; /* blk addr of cyl grp summary area */
        int32_t  fs_cssize;             /* cyl grp summary area size / bytes */
        int32_t  fs_cgsize;             /* cyl grp block size / bytes */
/* these fields are derived from the hardware */
        int32_t  fs_ntrak;              /* tracks per cylinder */
        int32_t  fs_nsect;              /* DEV_BSIZE sectors per track */
        int32_t  fs_spc;        /* DEV_BSIZE sectors per cylinder */
/* this comes from the disk driver partitioning */
        int32_t  fs_ncyl;               /* cylinders in file system */
/* these fields can be computed from the others */
        int32_t  fs_cpg;        /* cylinders per group */
        u_int32_t fs_ipg;               /* inodes per group */
        int32_t  fs_fpg;        /* blocks per group * fs_frag */
/* this data must be re-computed after crashes */
        struct   csum fs_ffs1_cstotal;  /* cylinder summary information */
/* these fields are cleared at mount time */
        int8_t   fs_fmod;               /* super block modified flag */
        int8_t   fs_clean;              /* file system is clean flag */
        int8_t   fs_ronly;              /* mounted read-only flag */
        int8_t   fs_ffs1_flags;         /* see FS_ below */
        u_char   fs_fsmnt[MAXMNTLEN];   /* name mounted on */
        u_char   fs_volname[MAXVOLLEN]; /* volume name */
        u_int64_t fs_swuid;             /* system-wide uid */
        int32_t  fs_pad;        /* due to alignment of fs_swuid */
/* these fields retain the current block allocation info */
        int32_t  fs_cgrotor;            /* last cg searched */
        void    *fs_ocsp[NOCSPTRS];     /* padding; was list of fs_cs buffers */
        u_int8_t *fs_contigdirs; /* # of contiguously allocated dirs */
        struct csum *fs_csp;            /* cg summary info buffer for fs_cs */
        int32_t *fs_maxcluster;         /* max cluster in each cyl group */
        u_char  *fs_active;             /* reserved for snapshots */
        int32_t  fs_cpc;        /* cyl per cycle in postbl */
/* this area is only allocated if fs_ffs1_flags & FS_FLAGS_UPDATED */
        int32_t  fs_maxbsize;          /* maximum blocking factor permitted */
        int64_t  fs_spareconf64[17];   /* old rotation block list head */
        int64_t  fs_sblockloc;         /* offset of standard super block */
        struct   csum_total fs_cstotal; /* cylinder summary information */
        int64_t  fs_time;              /* time last written */
        int64_t  fs_size;              /* number of blocks in fs */
        int64_t  fs_dsize;             /* number of data blocks in fs */
        int64_t  fs_csaddr;            /* blk addr of cyl grp summary area */
        int64_t  fs_pendingblocks;     /* blocks in process of being freed */
        u_int32_t fs_pendinginodes;   /* inodes in process of being freed */
        u_int32_t fs_snapinum[FSMAXSNAP];/* space reserved for snapshots */
/* back to stuff that has been around a while */
        u_int32_t fs_avgfilesize; /* expected average file size */
        u_int32_t fs_avgfpdir;          /* expected # of files per directory */
        int32_t  fs_sparecon[26];       /* reserved for future constants */
        u_int32_t fs_flags;             /* see FS_ flags below */
        int32_t  fs_fscktime;           /* last time fsck(8)ed */
        int32_t  fs_contigsumsize;      /* size of cluster summary array */
        int32_t  fs_maxsymlinklen;      /* max length of an internal symlink */
        int32_t  fs_inodefmt;           /* format of on-disk inodes */
        u_int64_t fs_maxfilesize; /* maximum representable file size */
        int64_t  fs_qbmask;             /* ~fs_bmask - for use with quad size */
        int64_t  fs_qfmask;             /* ~fs_fmask - for use with quad size */
        int32_t  fs_state;              /* validate fs_clean field */
        int32_t  fs_postblformat;       /* format of positional layout tables */
        int32_t  fs_nrpos;              /* number of rotational positions */
        int32_t  fs_postbloff;          /* (u_int16) rotation block list head */
        int32_t  fs_rotbloff;           /* (u_int8) blocks for each rotation */
        int32_t  fs_magic;              /* magic number */
        u_int8_t fs_space[1];           /* list of blocks for each rotation */
/* actually longer */
};

/*
 * Filesystem identification
 */
#define  FS_MAGIC 0x011954 /* the fast filesystem magic number */
#define  FS_UFS1_MAGIC    0x011954 /* the fast filesystem magic number */
#define  FS_UFS2_MAGIC    0x19540119      /* UFS fast filesystem magic number */
#define  FS_OKAY          0x7c269d38      /* superblock checksum */
#define FS_42INODEFMT     -1              /* 4.2BSD inode format */
#define FS_44INODEFMT     2               /* 4.4BSD inode format */
```

```
/*
 * Filesystem clean flags
 */
#define  FS_ISCLEAN      0x01
#define  FS_WASCLEAN     0x02

/*
 * Preference for optimization.
 */
#define FS_OPTTIME       0       /* minimize allocation time */
#define FS_OPTSPACE      1       /* minimize disk fragmentation */

/*
 * Filesystem flags.
 */
#define FS_UNCLEAN       0x01    /* filesystem not clean at mount */
#define FS_DOSOFTDEP     0x02    /* filesystem using soft dependencies */
/*
 * The following flag is used to detect a FFS1 file system that had its flags
 * moved to the new (FFS2) location for compatibility.
 */
#define FS_FLAGS_UPDATED 0x80    /* file system has FFS2-like flags */

/*
 * Rotational layout table format types
 */
#define FS_42POSTBLFMT          -1      /* 4.2BSD rotational table format */
#define FS_DYNAMICPOSTBLFMT      1      /* dynamic rotational table format */
/*
 * Macros for access to superblock array structures
 */
#define fs_rotbl(fs) \
    (((fs)->fs_postblformat == FS_42POSTBLFMT) \
    ? ((fs)->fs_space) \
    : ((u_int8_t *)((u_int8_t *)(fs) + (fs)->fs_rotbloff)))

/*
 * The size of a cylinder group is calculated by CGSIZE. The maximum size
 * is limited by the fact that cylinder groups are at most one block.
 * Its size is derived from the size of the maps maintained in the
 * cylinder group and the (struct cg) size.
 */
#define CGSIZE(fs) \
    /* base cg */        (sizeof(struct cg) + sizeof(int32_t) + \
    /* blktot size */    (fs)->fs_cpg * sizeof(int32_t) + \
    /* blks size */      (fs)->fs_cpg * (fs)->fs_nrpos * sizeof(int16_t) + \
    /* inode map */      howmany((fs)->fs_ipg, NBBY) + \
    /* block map */      howmany((fs)->fs_fpg, NBBY) + \
    /* if present */     ((fs)->fs_contigsumsize <= 0 ? 0 : \
    /* cluster sum */    (fs)->fs_contigsumsize * sizeof(int32_t) + \
    /* cluster map */    howmany(fragstoblks(fs, (fs)->fs_fpg), NBBY)))

/*
 * Convert cylinder group to base address of its global summary info.
 */
#define fs_cs(fs, indx) fs_csp[indx]

/*
 * Cylinder group block for a file system.
 */
#define  CG_MAGIC 0x090255
struct cg {
        int32_t   cg_firstfield;          /* historic cyl groups linked list */
        int32_t   cg_magic;               /* magic number */
        int32_t   cg_time;                /* time last written */
        u_int32_t cg_cgx;                 /* we are the cgx'th cylinder group */
        int16_t   cg_ncyl;                /* number of cyl's this cg */
        int16_t   cg_niblk;               /* number of inode blocks this cg */
        u_int32_t cg_ndblk;               /* number of data blocks this cg */
        struct   csum cg_cs;              /* cylinder summary information */
        u_int32_t cg_rotor;               /* position of last used block */
        u_int32_t cg_frotor;              /* position of last used frag */
        u_int32_t cg_irotor;              /* position of last used inode */
        u_int32_t cg_frsum[MAXFRAG];      /* counts of available frags */
        int32_t   cg_btotoff;             /* (int32) block totals per cylinder */
        int32_t   cg_boff;                /* (u_int16) free block positions */
        u_int32_t cg_iusedoff;            /* (u_int8) used inode map */
        u_int32_t cg_freeoff;             /* (u_int8) free block map */
        u_int32_t cg_nextfreeoff;/* (u_int8) next available space */
        u_int32_t cg_clustersumoff;       /* (u_int32) counts of avail clusters */
        u_int32_t cg_clusteroff; /* (u_int8) free cluster map */
        u_int32_t cg_nclusterblks;        /* number of clusters this cg */
        u_int32_t cg_ffs2_niblk; /* number of inode blocks this cg */
        u_int32_t cg_initediblk; /* last initialized inode */
        int32_t   cg_sparecon32[3];       /* reserved for future use */
        int64_t   cg_ffs2_time;           /* time last written */
        int64_t   cg_sparecon64[3];       /* reserved for future use */
/* actually longer */
};

/*
 * Macros for access to cylinder group array structures
 */
#define cg_blktot(cgp) \
    (((cgp)->cg_magic != CG_MAGIC) \
    ? (((struct ocg *)(cgp))->cg_btot) \
    : ((int32_t *)((u_int8_t *)(cgp) + (cgp)->cg_btotoff)))
#define cg_blks(fs, cgp, cylno) \
    (((cgp)->cg_magic != CG_MAGIC) \
    ? (((struct ocg *)(cgp))->cg_b[cylno]) \
    : ((int16_t *)((u_int8_t *)(cgp) + \
        (cgp)->cg_boff) + (cylno) * (fs)->fs_nrpos))
#define cg_inosused(cgp) \
    (((cgp)->cg_magic != CG_MAGIC) \
    ? (((struct ocg *)(cgp))->cg_iused) \
    : ((u_int8_t *)((u_int8_t *)(cgp) + (cgp)->cg_iusedoff)))
#define cg_blksfree(cgp) \
    (((cgp)->cg_magic != CG_MAGIC) \
    ? (((struct ocg *)(cgp))->cg_free) \
    : ((u_int8_t *)((u_int8_t *)(cgp) + (cgp)->cg_freeoff)))
#define cg_chkmagic(cgp) \
    ((cgp)->cg_magic == CG_MAGIC || ((struct ocg *)(cgp))->cg_magic == CG_MAGIC)
#define cg_clustersfree(cgp) \
    ((u_int8_t *)((u_int8_t *)(cgp) + (cgp)->cg_clusteroff))
#define cg_clustersum(cgp) \
    ((int32_t *)((u_int8_t *)(cgp) + (cgp)->cg_clustersumoff))

/*
 * The following structure is defined
 * for compatibility with old file systems.
 */
struct ocg {
        int32_t   cg_firstfield;          /* historic linked list of cyl groups */
        int32_t   cg_unused_1;            /*     used for incore cyl groups */
        int32_t   cg_time;                /* time last written */
        int32_t   cg_cgx;         /* we are the cgx'th cylinder group */
        int16_t   cg_ncyl;                /* number of cyl's this cg */
        int16_t   cg_niblk;               /* number of inode blocks this cg */
        int32_t   cg_ndblk;               /* number of data blocks this cg */
        struct   csum cg_cs;              /* cylinder summary information */
        int32_t   cg_rotor;               /* position of last used block */
        int32_t   cg_frotor;              /* position of last used frag */
        int32_t   cg_irotor;              /* position of last used inode */
        int32_t   cg_frsum[8];            /* counts of available frags */
        int32_t   cg_btot[32];            /* block totals per cylinder */
        int16_t   cg_b[32][8];            /* positions of free blocks */
        u_int8_t cg_iused[256];           /* used inode map */
        int32_t   cg_magic;               /* magic number */
        u_int8_t cg_free[1];              /* free block map */
/* actually longer */
};

/*
 * Turn file system block numbers into disk block addresses.
 * This maps file system blocks to DEV_BSIZE (a.k.a. 512-byte) size disk
 * blocks.
 */
#define fsbtodb(fs, b)   ((b) << (fs)->fs_fsbtodb)
#define dbtofsb(fs, b)   ((b) >> (fs)->fs_fsbtodb)

/*
 * Cylinder group macros to locate things in cylinder groups.
 * They calc file system addresses of cylinder group data structures.
 */
#define cgbase(fs, c)    ((daddr_t)(fs)->fs_fpg * (c))
#define cgdata(fs, c)    (cgdmin(fs, c) + (fs)->fs_minfree)        /* data zone */
#define cgmeta(fs, c)    (cgdmin(fs, c))                           /* meta data */
#define cgdmin(fs, c)    (cgstart(fs, c) + (fs)->fs_dblkno)        /* 1st data */
#define cgimin(fs, c)    (cgstart(fs, c) + (fs)->fs_iblkno)        /* inode blk */
#define cgsblock(fs, c)  (cgstart(fs, c) + (fs)->fs_sblkno)        /* super blk */
#define cgtod(fs, c)     (cgstart(fs, c) + (fs)->fs_cblkno)        /* cg block */
#define cgstart(fs, c)                                                     \
        (cgbase(fs, c) + (fs)->fs_cgoffset * ((c) & ~((fs)->fs_cgmask)))

/*
 * Macros for handling inode numbers:
 *     inode number to file system block offset.
 *     inode number to cylinder group number.
 *     inode number to file system block address.
 */
#define ino_to_cg(fs, x) ((x) / (fs)->fs_ipg)
#define ino_to_fsba(fs, x)                                                 \
        ((daddr_t)(cgimin(fs, ino_to_cg(fs, x)) +                  \
            (blkstofrags((fs), (((x) % (fs)->fs_ipg) / INOPB(fs))))))
#define ino_to_fsbo(fs, x)       ((x) % INOPB(fs))

/*
 * Give cylinder group number for a file system block.
 * Give frag block number in cylinder group for a file system block.
 */
#define dtog(fs, d)      ((d) / (fs)->fs_fpg)
#define dtogd(fs, d)     ((d) % (fs)->fs_fpg)

/*
 * Extract the bits for a block from a map.
 * Compute the cylinder and rotational position of a cyl block addr.
 */
#define blkmap(fs, map, loc) \
    (((map)[(loc) / NBBY] >> ((loc) % NBBY)) & (0xff >> (NBBY - (fs)->fs_frag)))
#define cbtocylno(fs, bno) \
    (fsbtodb(fs, bno) / (fs)->fs_spc)
#define cbtorpos(fs, bno) \
    ((fs)->fs_nrpos <= 1 ? 0 : \
     (fsbtodb(fs, bno) % (fs)->fs_spc / (fs)->fs_nsect * (fs)->fs_trackskew + \
     fsbtodb(fs, bno) % (fs)->fs_spc % (fs)->fs_nsect * (fs)->fs_interleave) % \
     (fs)->fs_nsect * (fs)->fs_nrpos / (fs)->fs_npsect)

/*
 * The following macros optimize certain frequently calculated
 * quantities by using shifts and masks in place of divisions
 * modulos and multiplications.
 */
#define blkoff(fs, loc)          /* calculates (loc % fs->fs_bsize) */ \
        ((loc) & (fs)->fs_qbmask)
#define fragoff(fs, loc) /* calculates (loc % fs->fs_fsize) */ \
        ((loc) & (fs)->fs_qfmask)
#define lblktosize(fs, blk)      /* calculates ((off_t)blk * fs->fs_bsize) */ \
        ((off_t)(blk) << (fs)->fs_bshift)
#define lblkno(fs, loc)          /* calculates (loc / fs->fs_bsize) */ \
        ((loc) >> (fs)->fs_bshift)
#define numfrags(fs, loc)/* calculates (loc / fs->fs_fsize) */ \
        ((loc) >> (fs)->fs_fshift)
#define blkroundup(fs, size)     /* calculates roundup(size, fs->fs_bsize) */ \
        (((size) + (fs)->fs_qbmask) & (fs)->fs_bmask)
#define fragroundup(fs, size)    /* calculates roundup(size, fs->fs_fsize) */ \
        (((size) + (fs)->fs_qfmask) & (fs)->fs_fmask)
#define fragstoblks(fs, frags)   /* calculates (frags / fs->fs_frag) */ \
        ((frags) >> (fs)->fs_fragshift)
#define blkstofrags(fs, blks)    /* calculates (blks * fs->fs_frag) */ \
        ((blks) << (fs)->fs_fragshift)
#define fragnum(fs, fsb) /* calculates (fsb % fs->fs_frag) */ \
        ((fsb) & ((fs)->fs_frag - 1))
#define blknum(fs, fsb)          /* calculates rounddown(fsb, fs->fs_frag) */ \
        ((fsb) &~ ((fs)->fs_frag - 1))

/*
 * Determine the number of available frags given a
 * percentage to hold in reserve.
 */
#define freespace(fs, percentreserved) \
        (blkstofrags((fs), (fs)->fs_cstotal.cs_nbfree) + \
        (fs)->fs_cstotal.cs_nffree - ((fs)->fs_dsize * (percentreserved) / 100))

/*
 * Determining the size of a file block in the file system.
 */
#define blksize(fs, ip, lbn) \
        (((lbn) >= NDADDR || DIP((ip), size) >= ((lbn) + 1) << (fs)->fs_bshift) \
            ? (u_int64_t)(fs)->fs_bsize \
            : (fragroundup(fs, blkoff(fs, DIP((ip), size)))))
#define dblksize(fs, dip, lbn) \
        (((lbn) >= NDADDR || (dip)->di_size >= ((lbn) + 1) << (fs)->fs_bshift) \
            ? (u_int64_t)(fs)->fs_bsize \
            : (fragroundup(fs, blkoff(fs, (dip)->di_size))))

#define sblksize(fs, size, lbn) \
        (((lbn) >= NDADDR || (size) >= ((lbn) + 1) << (fs)->fs_bshift) \
            ? (u_int64_t)(fs)->fs_bsize \
            : (fragroundup(fs, blkoff(fs, (size)))))


/*
 * Number of disk sectors per block/fragment; assumes DEV_BSIZE byte
 * sector size.
 */
#define NSPB(fs) ((fs)->fs_nspf << (fs)->fs_fragshift)
#define NSPF(fs) ((fs)->fs_nspf)

/* Number of inodes per file system block (fs->fs_bsize) */
#define INOPB(fs)        ((fs)->fs_inopb)
/* Number of inodes per file system fragment (fs->fs_fsize) */
#define INOPF(fs)        ((fs)->fs_inopb >> (fs)->fs_fragshift)

/*
 * Number of indirects in a file system block.
 */
#define NINDIR(fs)       ((fs)->fs_nindir)

/* Maximum file size the kernel allows.
 * Even though ffs can handle files up to 16TB, we do limit the max file
 * to 2^31 pages to prevent overflow of a 32-bit unsigned int.  The buffer
 * cache has its own checks but a little added paranoia never hurts.
 */
#define FS_KERNMAXFILESIZE(pgsiz, fs)     ((u_int64_t)0x80000000 * \
    MIN((pgsiz), (fs)->fs_bsize) - 1)

extern const int inside[], around[];
extern const u_char *fragtbl[];
File: softdep.h


/*      $OpenBSD: softdep.h,v 1.18 2018/04/01 12:02:00 dhill Exp $ */

/*
 * Copyright 1998, 2000 Marshall Kirk McKusick. All Rights Reserved.
 *
 * The soft updates code is derived from the appendix of a University
 * of Michigan technical report (Gregory R. Ganger and Yale N. Patt,
 * "Soft Updates: A Solution to the Metadata Update Problem in File
 * Systems", CSE-TR-254-95, August 1995).
 *
 * Further information about soft updates can be obtained from:
 *
 *      Marshall Kirk McKusick           http://www.mckusick.com/softdep/
 *      1614 Oxford Street               mckusick@mckusick.com
 *      Berkeley, CA 94709-1608          +1-510-843-9542
 *      USA
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions
 * are met:
 *
 * 1. Redistributions of source code must retain the above copyright
```

```c
 *    notice, this list of conditions and the following disclaimer.
 * 2. Redistributions in binary form must reproduce the above copyright
 *    notice, this list of conditions and the following disclaimer in the
 *    documentation and/or other materials provided with the distribution.
 *
 * THIS SOFTWARE IS PROVIDED BY MARSHALL KIRK MCKUSICK ``AS IS'' AND ANY
 * EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED
 * WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE
 * DISCLAIMED.  IN NO EVENT SHALL MARSHALL KIRK MCKUSICK BE LIABLE FOR
 * ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
 * DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
 * OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
 * HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
 * LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
 * OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
 * SUCH DAMAGE.
 *
 *	@(#)softdep.h	9.7 (McKusick) 6/21/00
 * $FreeBSD: src/sys/ufs/ffs/softdep.h,v 1.10 2000/06/22 00:29:53 mckusick Exp $
 */

#include <sys/queue.h>

/*
 * Allocation dependencies are handled with undo/redo on the in-memory
 * copy of the data. A particular data dependency is eliminated when
 * it is ALLCOMPLETE: that is ATTACHED, DEPCOMPLETE, and COMPLETE.
 *
 * ATTACHED means that the data is not currently being written to
 * disk. UNDONE means that the data has been rolled back to a safe
 * state for writing to the disk. When the I/O completes, the data is
 * restored to its current form and the state reverts to ATTACHED.
 * The data must be locked throughout the rollback, I/O, and roll
 * forward so that the rolled back information is never visible to
 * user processes. The COMPLETE flag indicates that the item has been
 * written. For example, a dependency that requires that an inode be
 * written will be marked COMPLETE after the inode has been written
 * to disk. The DEPCOMPLETE flag indicates the completion of any other
 * dependencies such as the writing of a cylinder group map has been
 * completed. A dependency structure may be freed only when both it
 * and its dependencies have completed and any rollbacks that are in
 * progress have finished as indicated by the set of ALLCOMPLETE flags
 * all being set. The two MKDIR flags indicate additional dependencies
 * that must be done when creating a new directory. MKDIR_BODY is
 * cleared when the directory data block containing the "." and ".."
 * entries has been written. MKDIR_PARENT is cleared when the parent
 * inode with the increased link count for ".." has been written. When
 * both MKDIR flags have been cleared, the DEPCOMPLETE flag is set to
 * indicate that the directory dependencies have been completed. The
 * writing of the directory inode itself sets the COMPLETE flag which
 * then allows the directory entry for the new directory to be written
 * to disk. The RMDIR flag marks a dirrem structure as representing
 * the removal of a directory rather than a file. When the removal
 * dependencies are completed, additional work needs to be done
 * (truncation of the "." and ".." entries, an additional decrement
 * of the associated inode, and a decrement of the parent inode). The
 * DIRCHG flag marks a diradd structure as representing the changing
 * of an existing entry rather than the addition of a new one. When
 * the update is complete the dirrem associated with the inode for
 * the old name must be added to the worklist to do the necessary
 * reference count decrement. The GOINGAWAY flag indicates that the
 * data structure is frozen from further change until its dependencies
 * have been completed and its resources freed after which it will be
 * discarded. The IOSTARTED flag prevents multiple calls to the I/O
 * start routine from doing multiple rollbacks. The SPACECOUNTED flag
 * says that the files space has been accounted to the pending free
 * space count. The NEWBLOCK flag marks pagedep structures that have
 * just been allocated, so must be claimed by the inode before all
 * dependencies are complete. The ONWORKLIST flag shows whether the
 * structure is currently linked onto a worklist.
 *
 */
#define	ATTACHED	0x0001
#define	UNDONE		0x0002
#define	COMPLETE	0x0004
#define	DEPCOMPLETE	0x0008
#define	MKDIR_PARENT	0x0010	/* diradd & mkdir only */
#define	MKDIR_BODY	0x0020	/* diradd & mkdir only */
#define	RMDIR		0x0040	/* dirrem only */
#define	DIRCHG		0x0080	/* diradd & dirrem only */
#define	GOINGAWAY	0x0100	/* indirdep only */
#define	IOSTARTED	0x0200	/* inodedep & pagedep only */
#define	SPACECOUNTED	0x0400	/* inodedep only */
#define	NEWBLOCK	0x0800	/* pagedep only */
#define	UFS1FMT		0x2000	/* indirdep only */
#define	ONWORKLIST	0x8000

#define	ALLCOMPLETE	(ATTACHED | COMPLETE | DEPCOMPLETE)

#define	DEP_BITS "\020\01ATTACHED\02UNDONE\03COMPLETE\04DEPCOMPLETE" \
    "\05MKDIR_PARENT\06MKDIR_BODY\07RMDIR\010DIRCHG\011GOINGAWAY" \
    "\012IOSTARTED\013SPACECOUNTED\014NEWBLOCK\016UFS1FMT\020ONWORKLIST"

/*
 * The workitem queue.
 *
 * It is sometimes useful and/or necessary to clean up certain dependencies
 * in the background rather than during execution of an application process
 * or interrupt service routine. To realize this, we append dependency
 * structures corresponding to such tasks to a "workitem" queue. In a soft
 * updates implementation, most pending workitems should not wait for more
 * than a couple of seconds, so the filesystem syncer process awakens once
 * per second to process the items on the queue.
 */

/* LIST_HEAD(workhead, worklist); -- declared in buf.h */

/*
 * Each request can be linked onto a work queue through its worklist structure.
 * To avoid the need for a pointer to the structure itself, this structure
 * MUST be declared FIRST in each type in which it appears! If more than one
 * worklist is needed in the structure, then a wk_data field must be added
 * and the macros below changed to use it.
 */
struct worklist {
	LIST_ENTRY(worklist)	wk_list;	/* list of work requests */
	unsigned short		wk_type;	/* type of request */
	unsigned short		wk_state;	/* state flags */
};
#define WK_DATA(wk) ((void *)(wk))
#define WK_PAGEDEP(wk) ((struct pagedep *)(wk))
#define WK_INODEDEP(wk) ((struct inodedep *)(wk))
#define WK_NEWBLK(wk) ((struct newblk *)(wk))
#define WK_BMSAFEMAP(wk) ((struct bmsafemap *)(wk))
#define WK_ALLOCDIRECT(wk) ((struct allocdirect *)(wk))
#define WK_INDIRDEP(wk) ((struct indirdep *)(wk))
#define WK_ALLOCINDIR(wk) ((struct allocindir *)(wk))
#define WK_FREEFRAG(wk) ((struct freefrag *)(wk))
#define WK_FREEBLKS(wk) ((struct freeblks *)(wk))
#define WK_FREEFILE(wk) ((struct freefile *)(wk))
#define WK_DIRADD(wk) ((struct diradd *)(wk))
#define WK_MKDIR(wk) ((struct mkdir *)(wk))
#define WK_DIRREM(wk) ((struct dirrem *)(wk))
#define WK_NEWDIRBLK(wk) ((struct newdirblk *)(wk))

/*
 * Various types of lists
 */
LIST_HEAD(dirremhd, dirrem);
LIST_HEAD(diraddhd, diradd);
LIST_HEAD(newblkhd, newblk);
LIST_HEAD(inodedephd, inodedep);
LIST_HEAD(allocindirhd, allocindir);
LIST_HEAD(allocdirecthd, allocdirect);
TAILQ_HEAD(allocdirectlst, allocdirect);

/*
 * The "pagedep" structure tracks the various dependencies related to
 * a particular directory page. If a directory page has any dependencies,
 * it will have a pagedep linked to its associated buffer. The
 * pd_dirremhd list holds the list of dirrem requests which decrement
 * inode reference counts. These requests are processed after the
 * directory page with the corresponding zero'ed entries has been
 * written. The pd_diraddhd list maintains the list of diradd requests
 * which cannot be committed until their corresponding inode has been
 * written to disk. Because a directory may have many new entries
 * being created, several lists are maintained hashed on bits of the
 * offset of the entry into the directory page to keep the lists from
 * getting too long. Once a new directory entry has been cleared to
 * be written, it is moved to the pd_pendinghd list. After the new
 * entry has been written to disk it is removed from the pd_pendinghd
 * list, any removed operations are done, and the dependency structure
 * is freed.
 */
#define DAHASHSZ 6
#define DIRADDHASH(offset) (((offset) >> 2) % DAHASHSZ)
struct pagedep {
	struct	worklist pd_list;	/* page buffer */
#	define	pd_state pd_list.wk_state /* check for multiple I/O starts */
	LIST_ENTRY(pagedep) pd_hash;	/* hashed lookup */
	struct	mount *pd_mnt;		/* associated mount point */
	ufsino_t pd_ino;		/* associated file */
	daddr_t pd_lbn;		/* block within file */
	struct	dirremhd pd_dirremhd;	/* dirrem's waiting for page */
	struct	diraddhd pd_diraddhd[DAHASHSZ]; /* diradd dir entry updates */
	struct	diraddhd pd_pendinghd;	/* directory entries awaiting write */
};

/*
 * The "inodedep" structure tracks the set of dependencies associated
 * with an inode. One task that it must manage is delayed operations
 * (i.e., work requests that must be held until the inodedep's associated
 * inode has been written to disk). Getting an inode from its incore
 * state to the disk requires two steps to be taken by the filesystem
 * in this order: first the inode must be copied to its disk buffer by
 * the VOP_UPDATE operation; second the inode's buffer must be written
 * to disk. To ensure that both operations have happened in the required
 * order, the inodedep maintains two lists. Delayed operations are
 * placed on the id_inowait list. When the VOP_UPDATE is done, all
 * operations on the id_inowait list are moved to the id_bufwait list.
 * When the buffer is written, the items on the id_bufwait list can be
 * safely moved to the work queue to be processed. A second task of the
 * inodedep structure is to track the status of block allocation within
 * the inode.  Each block that is allocated is represented by an
 * "allocdirect" structure (see below). It is linked onto the id_newinoupdt
 * list until both its contents and its allocation in the cylinder
 * group map have been written to disk. Once these dependencies have been
 * satisfied, it is removed from the id_newinoupdt list and any followup
 * actions such as releasing the previous block or fragment are placed
 * on the id_inowait list. When an inode is updated (a VOP_UPDATE is
 * done), the "inodedep" structure is linked onto the buffer through
 * its worklist. Thus, it will be notified when the buffer is about
 * to be written and when it is done. At the update time, all the
 * elements on the id_newinoupdt list are moved to the id_inoupdt list
 * since those changes are now relevant to the copy of the inode in the
 * buffer. Also at update time, the tasks on the id_inowait list are
 * moved to the id_bufwait list so that they will be executed when
 * the updated inode has been written to disk. When the buffer containing
 * the inode is written to disk, any updates listed on the id_inoupdt
 * list are rolled back as they are not yet safe. Following the write,
 * the changes are once again rolled forward and any actions on the
 * id_bufwait list are processed (since those actions are now safe).
 * The entries on the id_inoupdt and id_newinoupdt lists must be kept
 * sorted by logical block number to speed the calculation of the size
 * of the rolled back inode (see explanation in initiate_write_inodeblock).
 * When a directory entry is created, it is represented by a diradd.
 * The diradd is added to the id_inowait list as it cannot be safely
 * written to disk until the inode that it represents is on disk. After
 * the inode is written, the id_bufwait list is processed and the diradd
 * entries are moved to the id_pendinghd list where they remain until
 * the directory block containing the name has been written to disk.
 * The purpose of keeping the entries on the id_pendinghd list is so that
 * the softdep_fsync function can find and push the inode's directory
 * name(s) as part of the fsync operation for that file.
 */
struct inodedep {
	struct	worklist id_list;	/* buffer holding inode block */
#	define	id_state id_list.wk_state /* inode dependency state */
	LIST_ENTRY(inodedep) id_hash;	/* hashed lookup */
	struct	fs *id_fs;		/* associated filesystem */
	ufsino_t id_ino;		/* dependent inode */
	nlink_t	id_nlinkdelta;		/* saved effective link count */
	union { /* Saved UFS1/UFS2 dinode contents */
		struct ufs1_dinode *idu_savedino1;
		struct ufs2_dinode *idu_savedino2;
	} id_un;
	size_t	id_unsize;		/* size of dinode contents union */
	LIST_ENTRY(inodedep) id_deps;	/* bmsafemap's list of inodedep's */
	struct	buf *id_buf;		/* related bmsafemap (if pending) */
	off_t	id_savedsize;		/* file size saved during rollback */
	struct	workhead id_pendinghd;	/* entries awaiting directory write */
	struct	workhead id_bufwait;	/* operations after inode written */
	struct	workhead id_inowait;	/* operations waiting inode update */
	struct	allocdirectlst id_inoupdt; /* updates before inode written */
	struct	allocdirectlst id_newinoupdt; /* updates when inode written */
};

#define	id_savedino1	id_un.idu_savedino1
#define	id_savedino2	id_un.idu_savedino2

/*
 * A "newblk" structure is attached to a bmsafemap structure when a block
 * or fragment is allocated from a cylinder group. Its state is set to
 * DEPCOMPLETE when its cylinder group map is written. It is consumed by
 * an associated allocdirect or allocindir allocation which will attach
 * themselves to the bmsafemap structure if the newblk's DEPCOMPLETE flag
 * is not set (i.e., its cylinder group map has not been written).
 */
struct newblk {
	LIST_ENTRY(newblk) nb_hash;	/* hashed lookup */
	struct	fs *nb_fs;		/* associated filesystem */
	daddr_t nb_newblkno;		/* allocated block number */
	int	nb_state;		/* state of bitmap dependency */
	LIST_ENTRY(newblk) nb_deps;	/* bmsafemap's list of newblk's */
	struct	bmsafemap *nb_bmsafemap; /* associated bmsafemap */
};

/*
 * A "bmsafemap" structure maintains a list of dependency structures
 * that depend on the update of a particular cylinder group map.
 * It has lists for newblks, allocdirects, allocindirs, and inodedeps.
 * It is attached to the buffer of a cylinder group block when any of
 * these things are allocated from the cylinder group. It is freed
 * after the cylinder group map is written and the state of its
 * dependencies are updated with DEPCOMPLETE to indicate that it has
 * been processed.
 */
struct bmsafemap {
	struct	worklist sm_list;	/* cylgrp buffer */
	struct	buf *sm_buf;		/* associated buffer */
	struct	allocdirecthd sm_allocdirecthd; /* allocdirect deps */
	struct	allocindirhd sm_allocindirhd; /* allocindir deps */
	struct	inodedephd sm_inodedephd; /* inodedep deps */
	struct	newblkhd sm_newblkhd;	/* newblk deps */
};

/*
 * An "allocdirect" structure is attached to an "inodedep" when a new block
 * or fragment is allocated and pointed to by the inode described by
 * "inodedep". The worklist is linked to the buffer that holds the block.
 * When the block is first allocated, it is linked to the bmsafemap
 * structure associated with the buffer holding the cylinder group map
 * from which it was allocated. When the cylinder group map is written
 * to disk, ad_state has the DEPCOMPLETE flag set. When the block itself
 * is written, the COMPLETE flag is set. Once both the cylinder group map
 * and the data itself have been written, it is safe to write the inode
 * that claims the block. If there was a previous fragment that had been
 * allocated before the file was increased in size, the old fragment may
 * be freed once the inode claiming the new block is written to disk.
 * This ad_fragfree request is attached to the id_inowait list of the
 * associated inodedep (pointed to by ad_inodedep) for processing after
 * the inode is written. When a block is allocated to a directory, an
 * fsync of a file whose name is within that block must ensure not only
 * that the block containing the file name has been written, but also
 * that the on-disk inode references that block. When a new directory
 * block is created, we allocate a newdirblk structure which is linked
 * to the associated allocdirect (on its ad_newdirblk list). When the
 * allocdirect has been satisfied, the newdirblk structure is moved to
 * the inodedep id_bufwait list of its directory to await the inode
 * being written. When the inode is written, the directory entries are
 * fully committed and can be deleted from their pagedep->id_pendinghd
 * and inodedep->id_pendinghd lists.
 */
struct allocdirect {
        struct  worklist ad_list;         /* buffer holding block */
#       define  ad_state ad_list.wk_state /* block pointer state */
        TAILQ_ENTRY(allocdirect) ad_next; /* inodedep's list of allocdirect's */
        daddr_t ad_lbn;          /* block within file */
        daddr_t ad_newblkno;              /* new value of block pointer */
        daddr_t ad_oldblkno;              /* old value of block pointer */
        long    ad_newsize;               /* size of new block */
        long    ad_oldsize;               /* size of old block */
        LIST_ENTRY(allocdirect) ad_deps; /* bmsafemap's list of allocdirect's */
        struct  buf *ad_buf;              /* cylgrp buffer (if pending) */
        struct  inodedep *ad_inodedep;    /* associated inodedep */
        struct  freefrag *ad_freefrag;    /* fragment to be freed (if any) */
        struct  workhead ad_newdirblk;    /* dir block to notify when written */
};

/*
 * A single "indirdep" structure manages all allocation dependencies for
 * pointers in an indirect block. The up-to-date state of the indirect
 * block is stored in ir_savedata. The set of pointers that may be safely
 * written to the disk is stored in ir_safecopy. The state field is used
 * only to track whether the buffer is currently being written (in which
 * case it is not safe to update ir_safecopy). Ir_deplisthd contains the
 * list of allocindir structures, one for each block that needs to be
 * written to disk. Once the block and its bitmap allocation have been
 * written the safecopy can be updated to reflect the allocation and the
 * allocindir structure freed. If ir_state indicates that an I/O on the
 * indirect block is in progress when ir_safecopy is to be updated, the
 * update is deferred by placing the allocindir on the ir_donehd list.
 * When the I/O on the indirect block completes, the entries on the
 * ir_donehd list are processed by updating their corresponding ir_safecopy
 * pointers and then freeing the allocindir structure.
 */
struct indirdep {
        struct  worklist ir_list;         /* buffer holding indirect block */
#       define  ir_state ir_list.wk_state /* indirect block pointer state */
        caddr_t ir_saveddata;             /* buffer cache contents */
        struct  buf *ir_savebp;           /* buffer holding safe copy */
        struct  allocindirhd ir_donehd;   /* done waiting to update safecopy */
        struct  allocindirhd ir_deplisthd; /* allocindir deps for this block */
};

/*
 * An "allocindir" structure is attached to an "indirdep" when a new block
 * is allocated and pointed to by the indirect block described by the
 * "indirdep". The worklist is linked to the buffer that holds the new block.
 * When the block is first allocated, it is linked to the bmsafemap
 * structure associated with the buffer holding the cylinder group map
 * from which it was allocated. When the cylinder group map is written
 * to disk, ai_state has the DEPCOMPLETE flag set. When the block itself
 * is written, the COMPLETE flag is set. Once both the cylinder group map
 * and the data itself have been written, it is safe to write the entry in
 * the indirect block that claims the block; the "allocindir" dependency
 * can then be freed as it is no longer applicable.
 */
struct allocindir {
        struct  worklist ai_list;         /* buffer holding indirect block */
#       define  ai_state ai_list.wk_state /* indirect block pointer state */
        LIST_ENTRY(allocindir) ai_next;   /* indirdep's list of allocindir's */
        int     ai_offset;                /* pointer offset in indirect block */
        daddr_t ai_newblkno;              /* new block pointer value */
        daddr_t ai_oldblkno;              /* old block pointer value */
        struct  freefrag *ai_freefrag;    /* block to be freed when complete */
        struct  indirdep *ai_indirdep;    /* address of associated indirdep */
        LIST_ENTRY(allocindir) ai_deps;   /* bmsafemap's list of allocindir's */
        struct  buf *ai_buf;              /* cylgrp buffer (if pending) */
};

/*
 * A "freefrag" structure is attached to an "inodedep" when a previously
 * allocated fragment is replaced with a larger fragment, rather than extended.
 * The "freefrag" structure is constructed and attached when the replacement
 * block is first allocated. It is processed after the inode claiming the
 * bigger block that replaces it has been written to disk. Note that the
 * ff_state field is used to store the uid, so may lose data. However,
 * the uid is used only in printing an error message, so is not critical.
 * Keeping it in a short keeps the data structure down to 32 bytes.
 */
struct freefrag {
        struct  worklist ff_list;         /* id_inowait or delayed worklist */
#       define  ff_state ff_list.wk_state /* owning user; should be uid_t */
        struct  vnode *ff_devvp; /* filesystem device vnode */
        struct  mount *ff_mnt;            /* associated mount point */
        daddr_t ff_blkno;                 /* fragment physical block number */
        long    ff_fragsize;              /* size of fragment being deleted */
        ufsino_t ff_inum; /* owning inode number */
};

/*
 * A "freeblks" structure is attached to an "inodedep" when the
 * corresponding file's length is reduced to zero. It records all
 * the information needed to free the blocks of a file after its
 * zero'ed inode has been written to disk.
 */
struct freeblks {
        struct  worklist fb_list;         /* id_inowait or delayed worklist */
#       define  fb_state fb_list.wk_state /* inode and dirty block state */
        ufsino_t fb_previousinum; /* inode of previous owner of blocks */
        struct  vnode *fb_devvp; /* filesystem device vnode */
        struct  mount *fb_mnt;            /* associated mount point */
        off_t   fb_oldsize;               /* previous file size */
        off_t   fb_newsize;               /* new file size */
        int     fb_chkcnt;                /* used to check cnt of blks released */
        uid_t   fb_uid;                   /* uid of previous owner of blocks */
        daddr_t fb_dblks[NDADDR]; /* direct blk ptrs to deallocate */
        daddr_t fb_iblks[NIADDR]; /* indirect blk ptrs to deallocate */
};

/*
 * A "freefile" structure is attached to an inode when its
 * link count is reduced to zero. It marks the inode as free in
 * the cylinder group map after the zero'ed inode has been written
 * to disk and any associated blocks and fragments have been freed.
 */
struct freefile {
        struct  worklist fx_list;         /* id_inowait or delayed worklist */
        mode_t  fx_mode;         /* mode of inode */
        ufsino_t fx_oldinum;     /* inum of the unlinked file */
        struct  vnode *fx_devvp; /* filesystem device vnode */
        struct  mount *fx_mnt;            /* associated mount point */
};

/*
 * A "diradd" structure is linked to an "inodedep" id_inowait list when a
 * new directory entry is allocated that references the inode described
 * by "inodedep". When the inode itself is written (either the initial
 * allocation for new inodes or with the increased link count for
 * existing inodes), the COMPLETE flag is set in da_state. If the entry
 * is for a newly allocated inode, the "inodedep" structure is associated
 * with a bmsafemap which prevents the inode from being written to disk
 * until the cylinder group has been updated. Thus the da_state COMPLETE
 * flag cannot be set until the inode bitmap dependency has been removed.
 * When creating a new file, it is safe to write the directory entry that
 * claims the inode once the referenced inode has been written. Since
 * writing the inode clears the bitmap dependencies, the DEPCOMPLETE flag
 * in the diradd can be set unconditionally when creating a file. When
 * creating a directory, there are two additional dependencies described by
 * mkdir structures (see their description below). When these dependencies
 * are resolved the DEPCOMPLETE flag is set in the diradd structure.
 * If there are multiple links created to the same inode, there will be
 * a separate diradd structure created for each link. The diradd is
 * linked onto the pg_diraddhd list of the pagedep for the directory
 * page that contains the entry. When a directory page is written,
 * the pg_diraddhd list is traversed to rollback any entries that are
 * not yet ready to be written to disk. If a directory entry is being
 * changed (by rename) rather than added, the DIRCHG flag is set and
 * the da_previous entry points to the entry that will be "removed"
 * once the new entry has been committed. During rollback, entries
 * with da_previous are replaced with the previous inode number rather
 * than zero.
 *
 * The overlaying of da_pagedep and da_previous is done to keep the
 * structure down to 32 bytes in size on a 32-bit machine. If a
 * da_previous entry is present, the pointer to its pagedep is available
 * in the associated dirrem entry. If the DIRCHG flag is set, the
 * da_previous entry is valid; if not set the da_pagedep entry is valid.
 * The DIRCHG flag never changes; it is set when the structure is created
 * if appropriate and is never cleared.
 */
struct diradd {
        struct  worklist da_list;         /* id_inowait or id_pendinghd list */
#       define  da_state da_list.wk_state /* state of the new directory entry */
        LIST_ENTRY(diradd) da_pdlist;     /* pagedep holding directory block */
        doff_t  da_offset;                /* offset of new dir entry in dir blk */
        ufsino_t da_newinum;     /* inode number for the new dir entry */
        union {
        struct  dirrem *dau_previous;     /* entry being replaced in dir change */
        struct  pagedep *dau_pagedep;     /* pagedep dependency for addition */
        } da_un;
};
#define da_previous da_un.dau_previous
#define da_pagedep da_un.dau_pagedep

/*
 * Two "mkdir" structures are needed to track the additional dependencies
 * associated with creating a new directory entry. Normally a directory
 * addition can be committed as soon as the newly referenced inode has been
 * written to disk with its increased link count. When a directory is
 * created there are two additional dependencies: writing the directory
 * data block containing the "." and ".." entries (MKDIR_BODY) and writing
 * the parent inode with the increased link count for ".." (MKDIR_PARENT).
 * These additional dependencies are tracked by two mkdir structures that
 * reference the associated "diradd" structure. When they have completed,
 * they set the DEPCOMPLETE flag on the diradd so that it knows that its
 * extra dependencies have been completed. The md_state field is used only
 * to identify which type of dependency the mkdir structure is tracking.
 * It is not used in the mainline code for any purpose other than consistency
 * checking. All the mkdir structures in the system are linked together on
 * a list. This list is needed so that a diradd can find its associated
 * mkdir structures and deallocate them if it is prematurely freed (as for
 * example if a mkdir is immediately followed by a rmdir of the same directory).
 * Here, the free of the diradd must traverse the list to find the associated
 * mkdir structures that reference it. The deletion would be faster if the
 * diradd structure were simply augmented to have two pointers that referenced
 * the associated mkdir's. However, this would increase the size of the diradd
 * structure from 32 to 64-bits to speed a very infrequent operation.
 */
struct mkdir {
        struct  worklist md_list;         /* id_inowait or buffer holding dir */
#       define  md_state md_list.wk_state /* type: MKDIR_PARENT or MKDIR_BODY */
        struct  diradd *md_diradd;        /* associated diradd */
        struct  buf *md_buf;              /* MKDIR_BODY: buffer holding dir */
        LIST_ENTRY(mkdir) md_mkdirs;      /* list of all mkdirs */
};
LIST_HEAD(mkdirlist, mkdir) mkdirlisthd;

/*
 * A "dirrem" structure describes an operation to decrement the link
 * count on an inode. The dirrem structure is attached to the pg_dirremhd
 * list of the pagedep for the directory page that contains the entry.
 * It is processed after the directory page with the deleted entry has
 * been written to disk.
 *
 * The overlaying of dm_pagedep and dm_dirinum is done to keep the
 * structure down to 32 bytes in size on a 32-bit machine. It works
 * because they are never used concurrently.
 */
struct dirrem {
        struct  worklist dm_list;         /* delayed worklist */
#       define  dm_state dm_list.wk_state /* state of the old directory entry */
        LIST_ENTRY(dirrem) dm_next;       /* pagedep's list of dirrem's */
        struct  mount *dm_mnt;            /* associated mount point */
        ufsino_t dm_oldinum;     /* inum of the removed dir entry */
        union {
        struct  pagedep *dmu_pagedep;     /* pagedep dependency for remove */
        ufsino_t dmu_dirinum;    /* parent inode number (for rmdir) */
        } dm_un;
};
#define dm_pagedep dm_un.dmu_pagedep
#define dm_dirinum dm_un.dmu_dirinum


/*
 * A "newdirblk" structure tracks the progress of a newly allocated
 * directory block from its creation until it is claimed by its on-disk
 * inode. When a block is allocated to a directory, an fsync of a file
 * whose name is within that block must ensure not only that the block
 * containing the file name has been written, but also that the on-disk
 * inode references that block. When a new directory block is created,
 * we allocate a newdirblk structure which is linked to the associated
 * allocdirect (on its ad_newdirblk list). When the allocdirect has been
 * satisfied, the newdirblk structure is moved to the inodedep id_bufwait
 * list of its directory to await the inode being written. When the inode
 * is written, the directory entries are fully committed and can be
 * deleted from their pagedep->id_pendinghd and inodedep->id_pendinghd
 * lists. Note that we could track directory blocks allocated to indirect
 * blocks using a similar scheme with the allocindir structures. Rather
 * than adding this level of complexity, we simply write those newly
 * allocated indirect blocks synchronously as such allocations are rare.
 */
struct newdirblk {
        struct  worklist db_list;/* id_inowait or pg_newdirblk */
#       define  db_state db_list.wk_state /* unused */
        struct  pagedep *db_pagedep;/* associated pagedep */
};
```