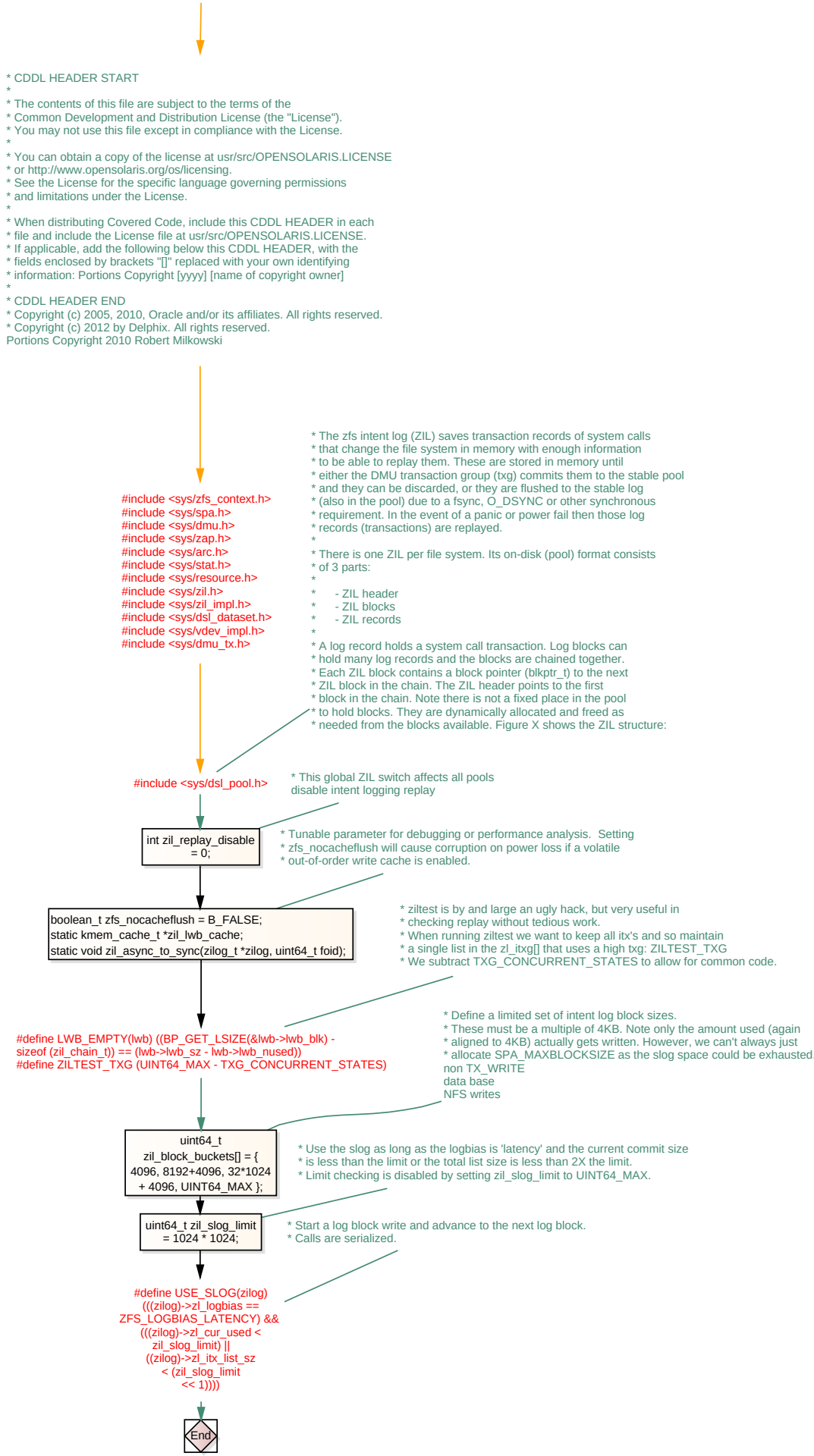
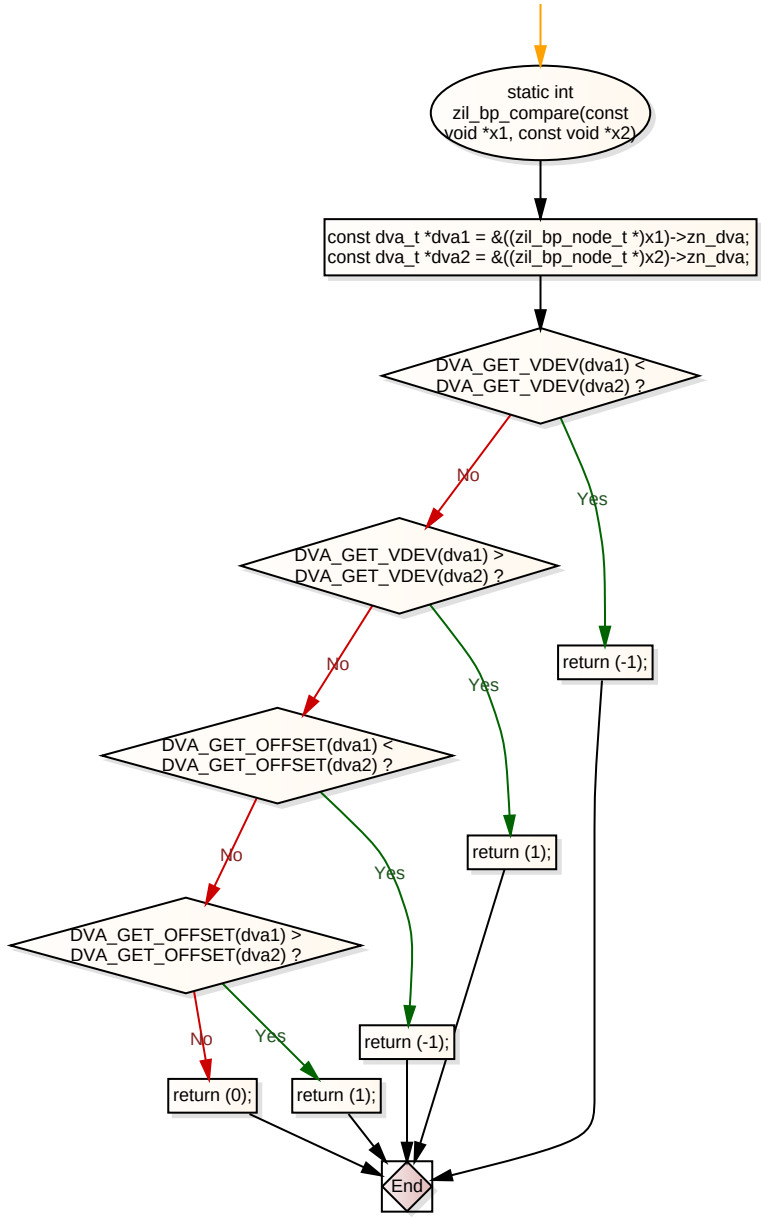
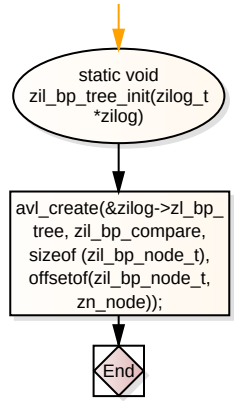
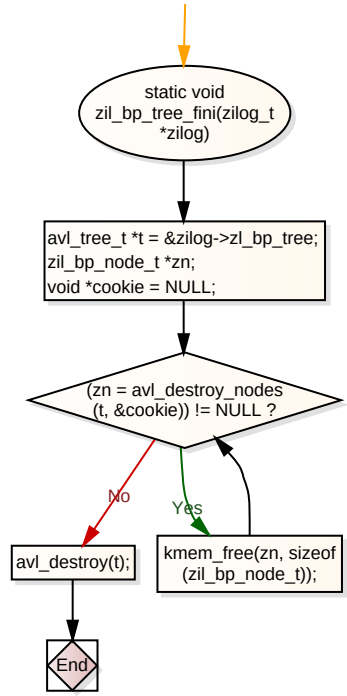
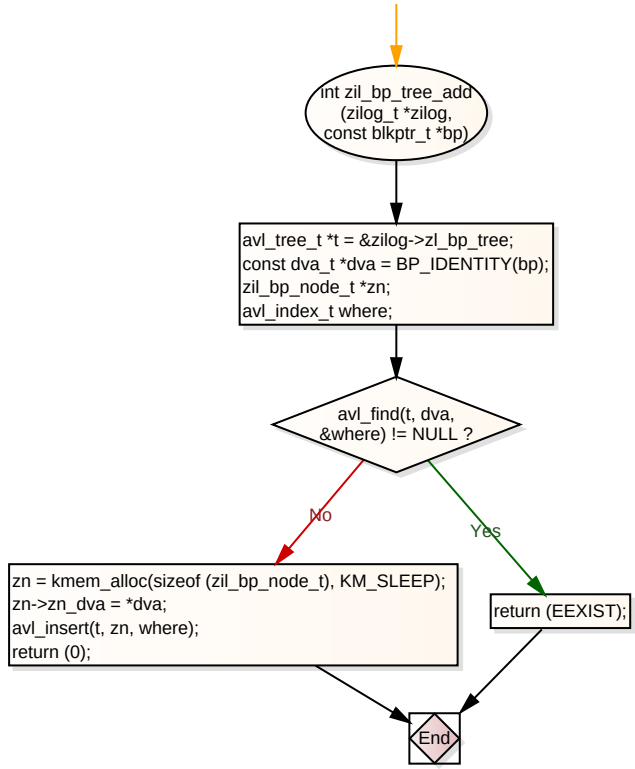
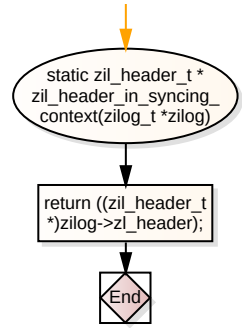

* CDDL HEADER START
*
* The contents of this file are subject to the terms of the
* Common Development and Distribution License (the "License").
* You may not use this file except in compliance with the License.
*
* You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
* or http://www.opensolaris.org/os/licensing.
* See the License for the specific language governing permissions
* and limitations under the License.
*
* When distributing Covered Code, include this CDDL HEADER in each
* file and include the License file at usr/src/OPENSOLARIS.LICENSE.
* If applicable, add the following below this CDDL HEADER, with the
* fields enclosed by brackets "[]" replaced with your own identifying
* information: Portions Copyright [yyyy] [name of copyright owner]
*
* CDDL HEADER END
* Copyright (c) 2005, 2010, Oracle and/or its affiliates. All rights reserved.
* Copyright (c) 2012 by Delphix. All rights reserved.
Portions Copyright 2010 Robert Milkowski
#include <sys/zfs_context.h>
#include <sys/spa.h>
#include <sys/dmu.h>
#include <sys/zap.h>
#include <sys/arc.h>
#include <sys/stat.h>
#include <sys/resource.h>
#include <sys/zil.h>
#include <sys/zil_impl.h>
#include <sys/dsl_dataset.h>
#include <sys/vdev_impl.h>
#include <sys/dmu_tx.h>
* The zfs intent log (ZIL) saves transaction records of system calls
* that change the file system in memory with enough information
* to be able to replay them. These are stored in memory until
* either the DMU transaction group (txg) commits them to the stable pool
* and they can be discarded, or they are flushed to the stable log
* (also in the pool) due to a fsync, O_DSYNC or other synchronous
* requirement. In the event of a panic or power fail then those log
* records (transactions) are replayed.
*
* There is one ZIL per file system. Its on-disk (pool) format consists
* of 3 parts:
*
*   - ZIL header
*   - ZIL blocks
*   - ZIL records
*
* A log record holds a system call transaction. Log blocks can
* hold many log records and the blocks are chained together.
* Each ZIL block contains a block pointer (blkptr_t) to the next
* ZIL block in the chain. The ZIL header points to the first
* block in the chain. Note there is not a fixed place in the pool
* to hold blocks. They are dynamically allocated and freed as
* needed from the blocks available. Figure X shows the ZIL structure:
#include <sys/dsl_pool.h>
* This global ZIL switch affects all pools
disable intent logging replay
int zil_replay_disable
= 0;
* Tunable parameter for debugging or performance analysis. Setting
* zfs_nocacheflush will cause corruption on power loss if a volatile
* out-of-order write cache is enabled.
boolean_t zfs_nocacheflush = B_FALSE;
static kmem_cache_t *zil_lwb_cache;
static void zil_async_to_sync(zilog_t *zilog, uint64_t foid);
* ziltest is by and large an ugly hack, but very useful in
* checking replay without tedious work.
* When running ziltest we want to keep all itx's and so maintain
* a single list in the zl_itxg[] that uses a high txg: ZILTEST_TXG
* We subtract TXG_CONCURRENT_STATES to allow for common code.
#define LWB_EMPTY(lwb) ((BP_GET_LSIZE(&lwb->lwb_blk) -
sizeof (zil_chain_t)) == (lwb->lwb_sz - lwb->lwb_nused))
#define ZILTEST_TXG (UINT64_MAX - TXG_CONCURRENT_STATES)
* Define a limited set of intent log block sizes.
* These must be a multiple of 4KB. Note only the amount used (again
* aligned to 4KB) actually gets written. However, we can't always just
* allocate SPA_MAXBLOCKSIZE as the slog space could be exhausted.
non TX_WRITE
data base
NFS writes
uint64_t
zil_block_buckets[] = {
4096, 8192+4096, 32*1024
+ 4096, UINT64_MAX };
* Use the slog as long as the logbias is 'latency' and the current commit size
* is less than the limit or the total list size is less than 2X the limit.
* Limit checking is disabled by setting zil_slog_limit to UINT64_MAX.
uint64_t zil_slog_limit
= 1024 * 1024;
* Start a log block write and advance to the next log block.
* Calls are serialized.
#define USE_SLOG(zilog)
(((zilog)->zl_logbias ==
ZFS_LOGBIAS_LATENCY) &&
(((zilog)->zl_cur_used <
zil_slog_limit) ||
((zilog)->zl_itx_list_sz
< (zil_slog_limit
<< 1))))
End

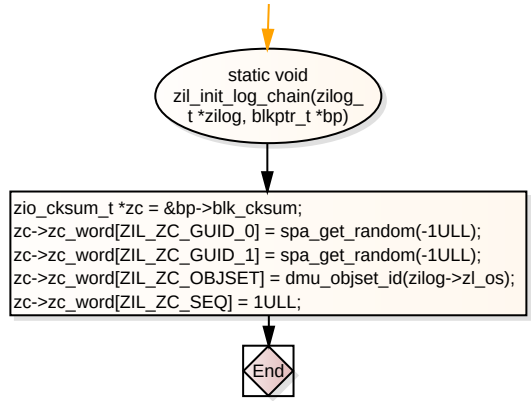

static int
zil_bp_compare(const
void *x1, const void *x2)
const dva_t *dva1 = &((zil_bp_node_t *)x1)->zn_dva;
const dva_t *dva2 = &((zil_bp_node_t *)x2)->zn_dva;
DVA_GET_VDEV(dva1) <
DVA_GET_VDEV(dva2) ?
No
Yes
DVA_GET_VDEV(dva1) >
DVA_GET_VDEV(dva2) ?
return (-1);
No
Yes
DVA_GET_OFFSET(dva1) <
DVA_GET_OFFSET(dva2) ?
return (1);
No
Yes
DVA_GET_OFFSET(dva1) >
DVA_GET_OFFSET(dva2) ?
return (-1);
No
Yes
return (0);
return (1);
End

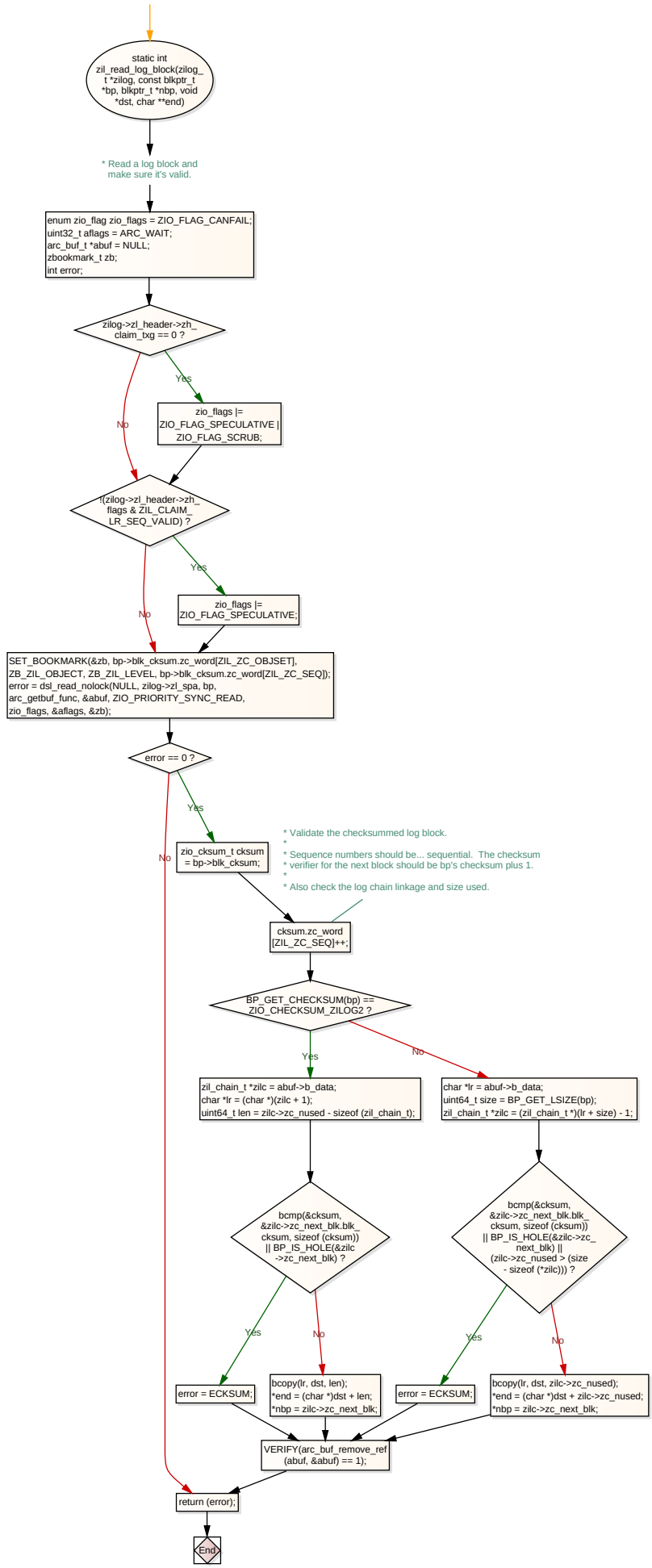

static void
zil_bp_tree_init(zilog_t
*zilog)
avl_create(&zilog->zl_bp_
tree, zil_bp_compare,
sizeof (zil_bp_node_t),
offsetof(zil_bp_node_t,
zn_node));
End

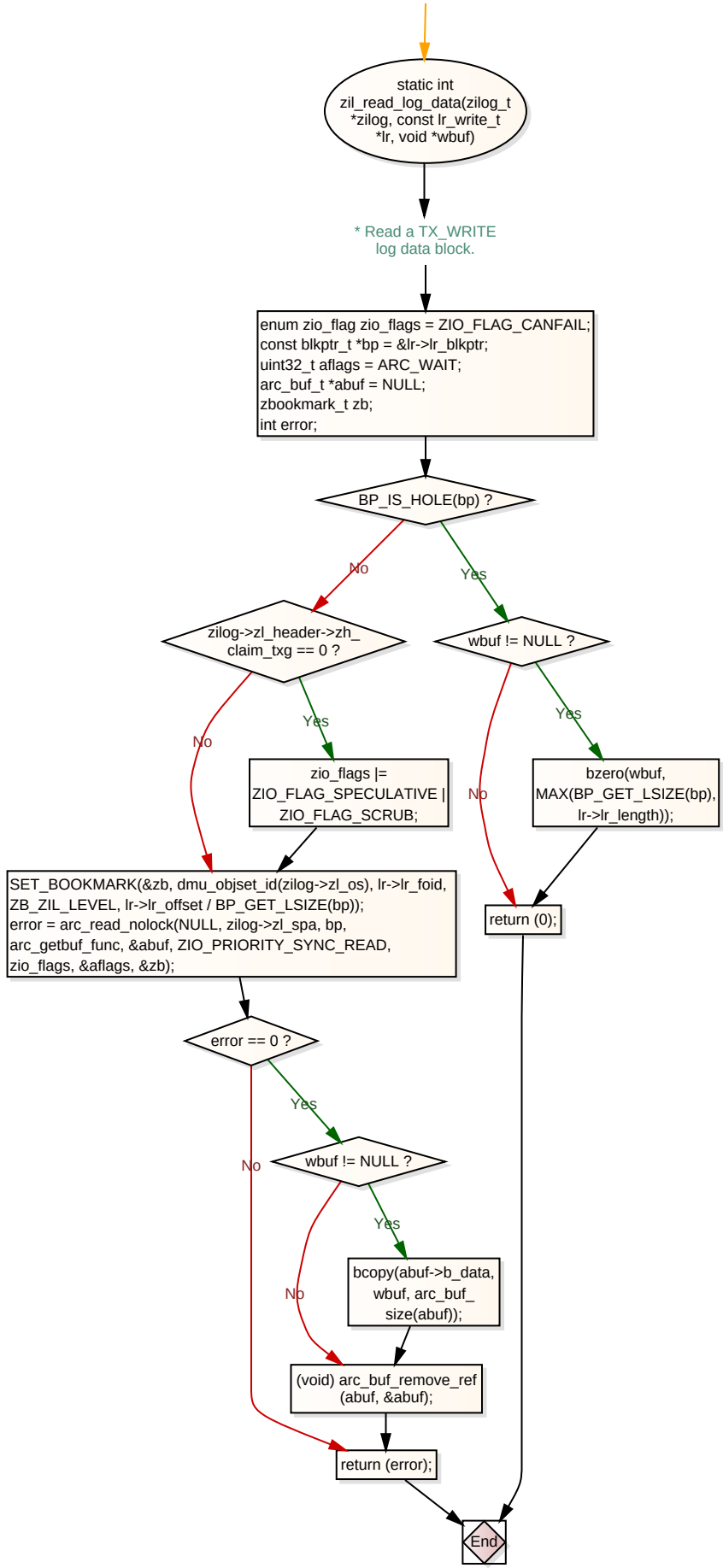

static void
zil_bp_tree_fini(zilog_t
*zilog)
avl_tree_t *t = &zilog->zl_bp_tree;
zil_bp_node_t *zn;
void *cookie = NULL;
(zn = avl_destroy_nodes
(t, &cookie)) != NULL ?
No
Yes
avl_destroy(t);
kmem_free(zn, sizeof
(zil_bp_node_t));
End

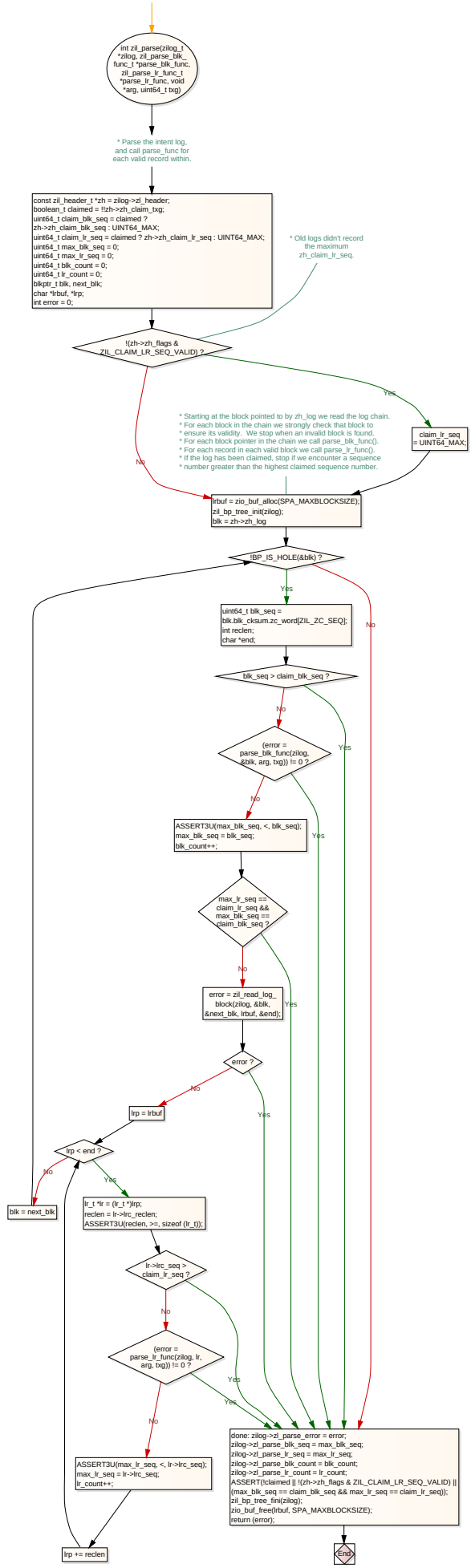

int zil_bp_tree_add
(zilog_t *zilog,
const blkptr_t *bp)
avl_tree_t *t = &zilog->zl_bp_tree;
const dva_t *dva = BP_IDENTITY(bp);
zil_bp_node_t *zn;
avl_index_t where;
avl_find(t, dva,
&where) != NULL ?
No
Yes
zn = kmem_alloc(sizeof (zil_bp_node_t), KM_SLEEP);
zn->zn_dva = *dva;
avl_insert(t, zn, where);
return (0);
return (EEXIST);
End

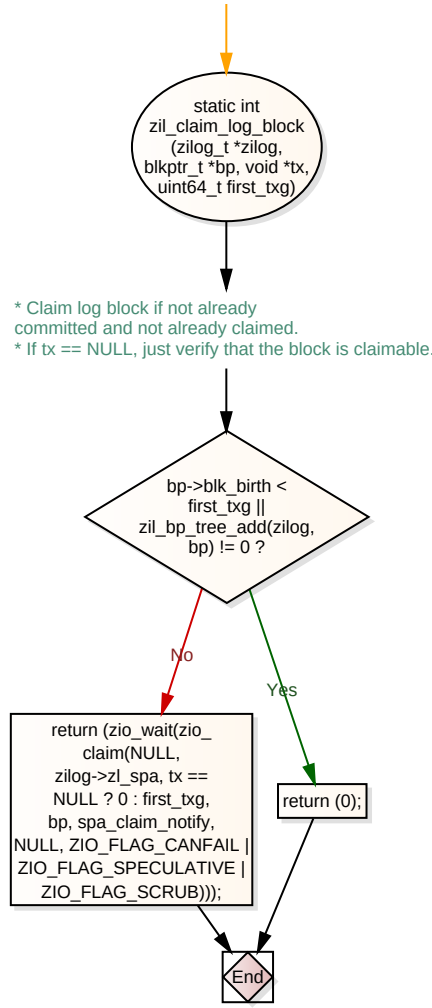

static zil_header_t *
zil_header_in_syncing_
context(zilog_t *zilog)
return ((zil_header_t
*)zilog->zl_header);
End

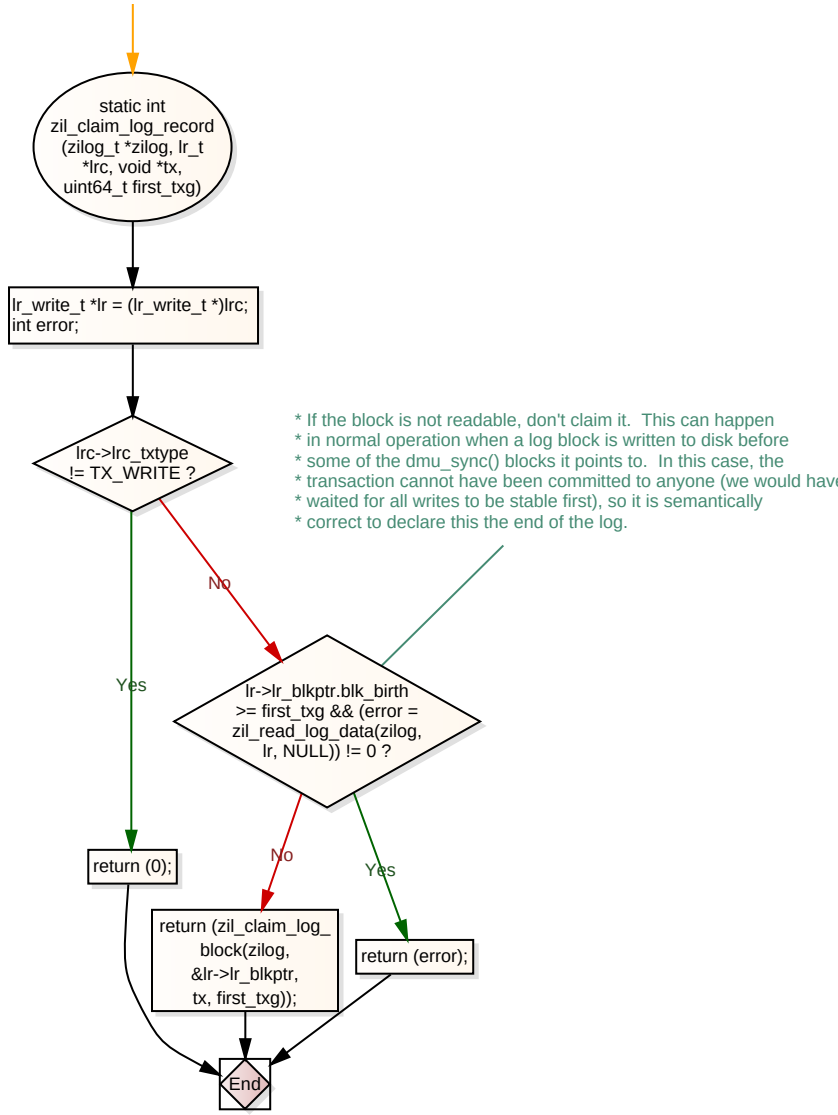

static void
zil_init_log_chain(zilog_
t *zilog, blkptr_t *bp)
zio_cksum_t *zc = &bp->blk_cksum;
zc->zc_word[ZIL_ZC_GUID_0] = spa_get_random(-1ULL);
zc->zc_word[ZIL_ZC_GUID_1] = spa_get_random(-1ULL);
zc->zc_word[ZIL_ZC_OBJSET] = dmu_objset_id(zilog->zl_os);
zc->zc_word[ZIL_ZC_SEQ] = 1ULL;
End

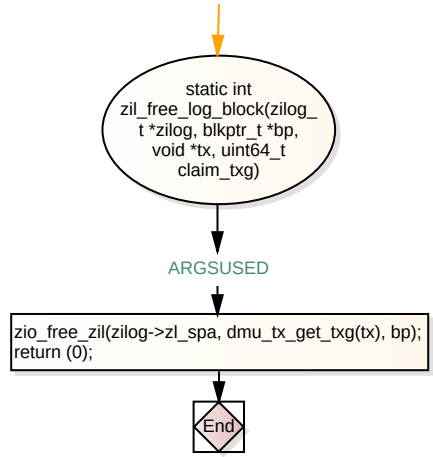

static int zil_read_log_block(zilog_t *zilog, const blkptr_t *bp, blkptr_t *nbp, void *dst, char **end)
* Read a log block and make sure it's valid.
enum zio_flag zio_flags = ZIO_FLAG_CANFAIL;
uint32_t aflags = ARC_WAIT;
arc_buf_t *abuf = NULL;
zbookmark_t zb;
int error;
zilog->zl_header->zh_claim_txg == 0 ?
Yes
No
zio_flags |= ZIO_FLAG_SPECULATIVE | ZIO_FLAG_SCRUB;
!(zilog->zl_header->zh_flags & ZIL_CLAIM_LR_SEQ_VALID) ?
Yes
No
zio_flags |= ZIO_FLAG_SPECULATIVE;
SET_BOOKMARK(&zb, bp->blk_cksum.zc_word[ZIL_ZC_OBJSET], ZB_ZIL_OBJECT, ZB_ZIL_LEVEL, bp->blk_cksum.zc_word[ZIL_ZC_SEQ]);
error = dsl_read_nolock(NULL, zilog->zl_spa, bp, arc_getbuf_func, &abuf, ZIO_PRIORITY_SYNC_READ, zio_flags, &aflags, &zb);
error == 0 ?
Yes
No
zio_cksum_t cksum = bp->blk_cksum;
* Validate the checksummed log block.
*
* Sequence numbers should be... sequential. The checksum
* verifier for the next block should be bp's checksum plus 1.
*
* Also check the log chain linkage and size used.
cksum.zc_word[ZIL_ZC_SEQ]++;
BP_GET_CHECKSUM(bp) == ZIO_CHECKSUM_ZILOG2 ?
Yes
No
zil_chain_t *zilc = abuf->b_data;
char *lr = (char *)(zilc + 1);
uint64_t len = zilc->zc_nused - sizeof (zil_chain_t);
char *lr = abuf->b_data;
uint64_t size = BP_GET_LSIZE(bp);
zil_chain_t *zilc = (zil_chain_t *)(lr + size) - 1;
bcmp(&cksum, &zilc->zc_next_blk.blk_cksum, sizeof (cksum)) || BP_IS_HOLE(&zilc->zc_next_blk) ?
bcmp(&cksum, &zilc->zc_next_blk.blk_cksum, sizeof (cksum)) || BP_IS_HOLE(&zilc->zc_next_blk) || (zilc->zc_nused > (size - sizeof (*zilc))) ?
Yes
No
Yes
No
error = ECKSUM;
bcopy(lr, dst, len);
*end = (char *)dst + len;
*nbp = zilc->zc_next_blk;
error = ECKSUM;
bcopy(lr, dst, zilc->zc_nused);
*end = (char *)dst + zilc->zc_nused;
*nbp = zilc->zc_next_blk;
VERIFY(arc_buf_remove_ref(abuf, &abuf) == 1);
return (error);
End

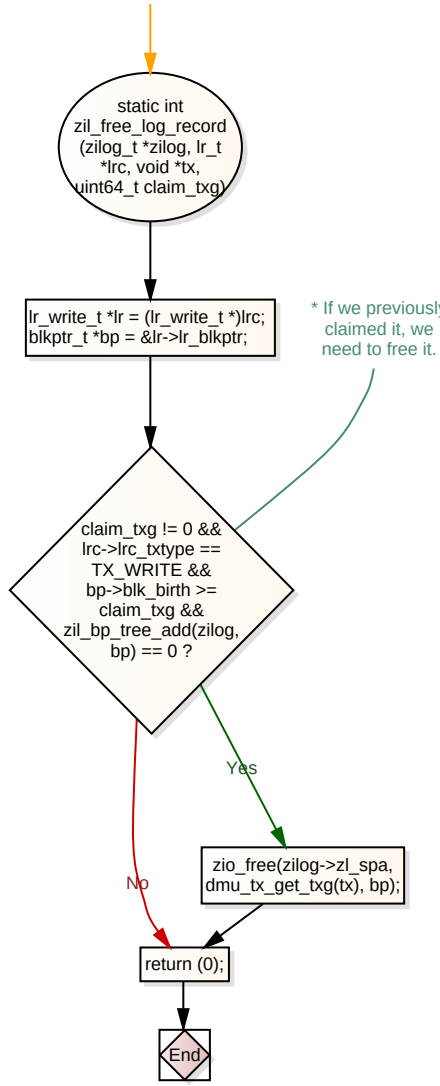

static int
zil_read_log_data(zilog_t
*zilog, const lr_write_t
*lr, void *wbuf)
* Read a TX_WRITE
log data block.
enum zio_flag zio_flags = ZIO_FLAG_CANFAIL;
const blkptr_t *bp = &lr->lr_blkptr;
uint32_t aflags = ARC_WAIT;
arc_buf_t *abuf = NULL;
zbookmark_t zb;
int error;
BP_IS_HOLE(bp) ?
No
Yes
zilog->zl_header->zh_
claim_txg == 0 ?
wbuf != NULL ?
No
Yes
No
Yes
zio_flags |=
ZIO_FLAG_SPECULATIVE |
ZIO_FLAG_SCRUB;
bzero(wbuf,
MAX(BP_GET_LSIZE(bp),
lr->lr_length));
SET_BOOKMARK(&zb, dmu_objset_id(zilog->zl_os), lr->lr_foid,
ZB_ZIL_LEVEL, lr->lr_offset / BP_GET_LSIZE(bp));
error = arc_read_nolock(NULL, zilog->zl_spa, bp,
arc_getbuf_func, &abuf, ZIO_PRIORITY_SYNC_READ,
zio_flags, &aflags, &zb);
return (0);
error == 0 ?
Yes
No
wbuf != NULL ?
Yes
No
bcopy(abuf->b_data,
wbuf, arc_buf_
size(abuf));
(void) arc_buf_remove_ref
(abuf, &abuf);
return (error);
End

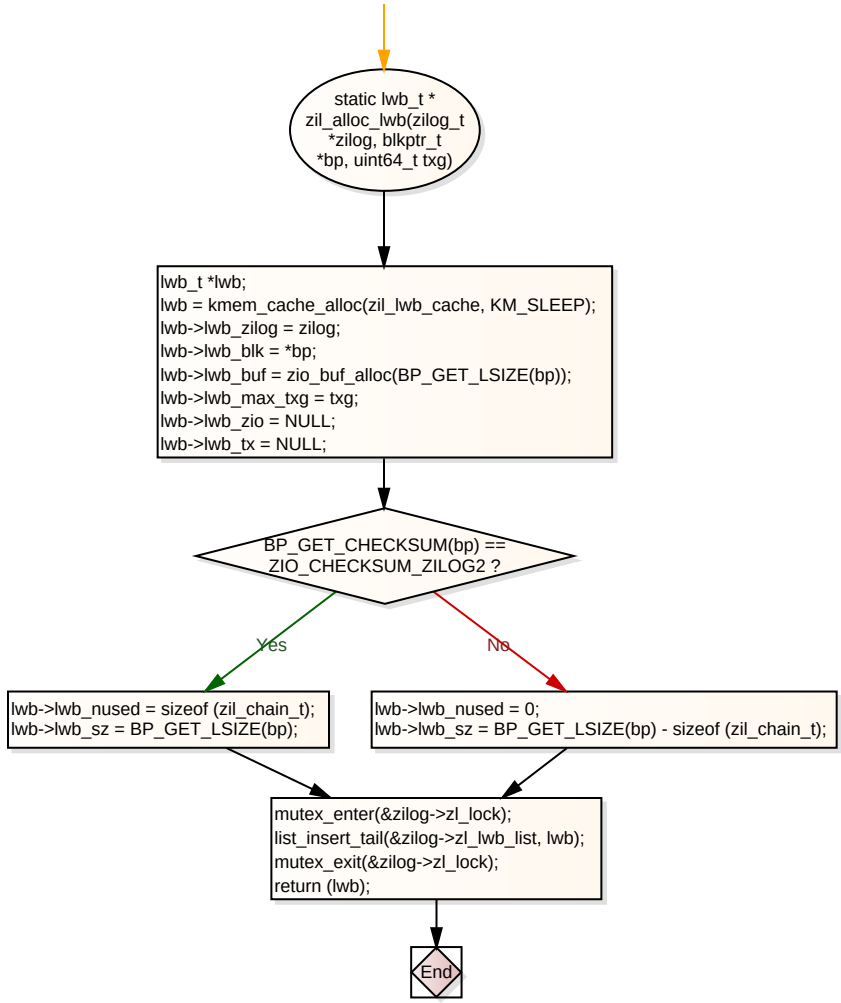

int zil_parse(zilog_t *zilog, zil_parse_blk_func_t *parse_blk_func, zil_parse_lr_func_t *parse_lr_func, void *arg, uint64_t txg)
* Parse the intent log, and call parse_func for each valid record within.
const zil_header_t *zh = zilog->zl_header;
boolean_t claimed = !!zh->zh_claim_txg;
uint64_t claim_blk_seq = claimed ? zh->zh_claim_blk_seq : UINT64_MAX;
uint64_t claim_lr_seq = claimed ? zh->zh_claim_lr_seq : UINT64_MAX;
uint64_t max_blk_seq = 0;
uint64_t max_lr_seq = 0;
uint64_t blk_count = 0;
uint64_t lr_count = 0;
blkptr_t blk, next_blk;
char *lrbuf, *lrp;
int error = 0;
* Old logs didn't record the maximum zh_claim_lr_seq.
!(zh->zh_flags & ZIL_CLAIM_LR_SEQ_VALID) ?
Yes
No
* Starting at the block pointed to by zh_log we read the log chain.
* For each block in the chain we strongly check that block to
* ensure its validity. We stop when an invalid block is found.
* For each block pointer in the chain we call parse_blk_func().
* For each record in each valid block we call parse_lr_func().
* If the log has been claimed, stop if we encounter a sequence
* number greater than the highest claimed sequence number.
claim_lr_seq = UINT64_MAX;
lrbuf = zio_buf_alloc(SPA_MAXBLOCKSIZE);
zil_bp_tree_init(zilog);
blk = zh->zh_log
!BP_IS_HOLE(&blk) ?
Yes
No
uint64_t blk_seq = blk.blk_cksum.zc_word[ZIL_ZC_SEQ];
int reclen;
char *end;
blk_seq > claim_blk_seq ?
No
Yes
(error = parse_blk_func(zilog, &blk, arg, txg)) != 0 ?
No
Yes
ASSERT3U(max_blk_seq, <, blk_seq);
max_blk_seq = blk_seq;
blk_count++;
Yes
max_lr_seq == claim_lr_seq && max_blk_seq == claim_blk_seq ?
No
Yes
error = zil_read_log_block(zilog, &blk, &next_blk, lrbuf, &end);
error ?
No
Yes
lrp = lrbuf
lrp < end ?
No
Yes
blk = next_blk
lr_t *lr = (lr_t *)lrp;
reclen = lr->lrc_reclen;
ASSERT3U(reclen, >=, sizeof (lr_t));
lr->lrc_seq > claim_lr_seq ?
No
Yes
(error = parse_lr_func(zilog, lr, arg, txg)) != 0 ?
No
Yes
Yes
ASSERT3U(max_lr_seq, <, lr->lrc_seq);
max_lr_seq = lr->lrc_seq;
lr_count++;
lrp += reclen
done: zilog->zl_parse_error = error;
zilog->zl_parse_blk_seq = max_blk_seq;
zilog->zl_parse_lr_seq = max_lr_seq;
zilog->zl_parse_blk_count = blk_count;
zilog->zl_parse_lr_count = lr_count;
ASSERT(!claimed || !(zh->zh_flags & ZIL_CLAIM_LR_SEQ_VALID) || (max_blk_seq == claim_blk_seq && max_lr_seq == claim_lr_seq));
zil_bp_tree_fini(zilog);
zio_buf_free(lrbuf, SPA_MAXBLOCKSIZE);
return (error);
End

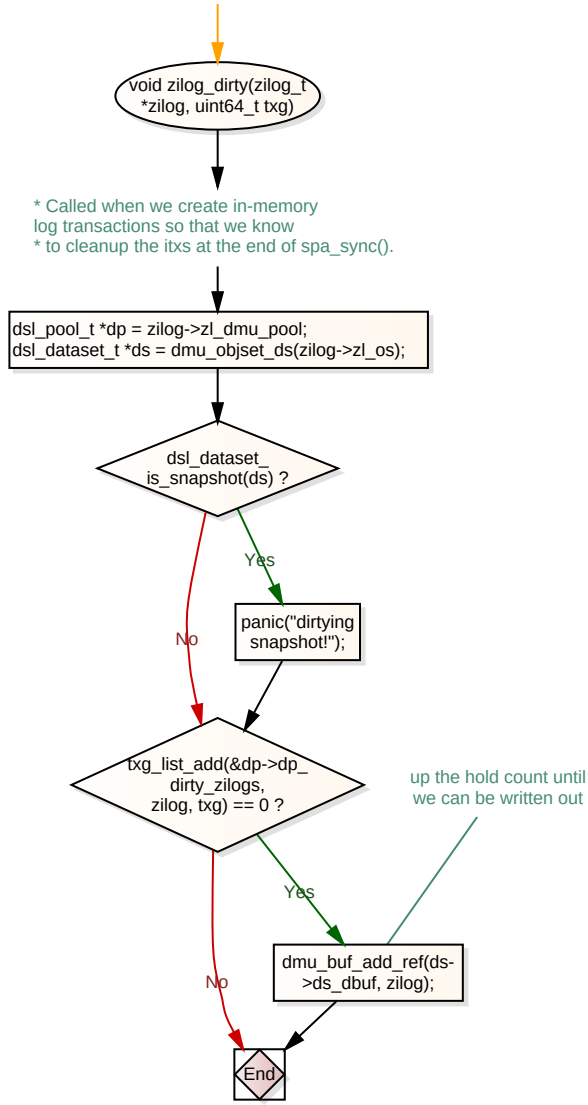

static int
zil_claim_log_block
(zilog_t *zilog,
blkptr_t *bp, void *tx,
uint64_t first_txg)
* Claim log block if not already
committed and not already claimed.
* If tx == NULL, just verify that the block is claimable.
bp->blk_birth <
first_txg ||
zil_bp_tree_add(zilog,
bp) != 0 ?
No
Yes
return (zio_wait(zio_
claim(NULL,
zilog->zl_spa, tx ==
NULL ? 0 : first_txg,
bp, spa_claim_notify,
NULL, ZIO_FLAG_CANFAIL |
ZIO_FLAG_SPECULATIVE |
ZIO_FLAG_SCRUB)));
return (0);
End

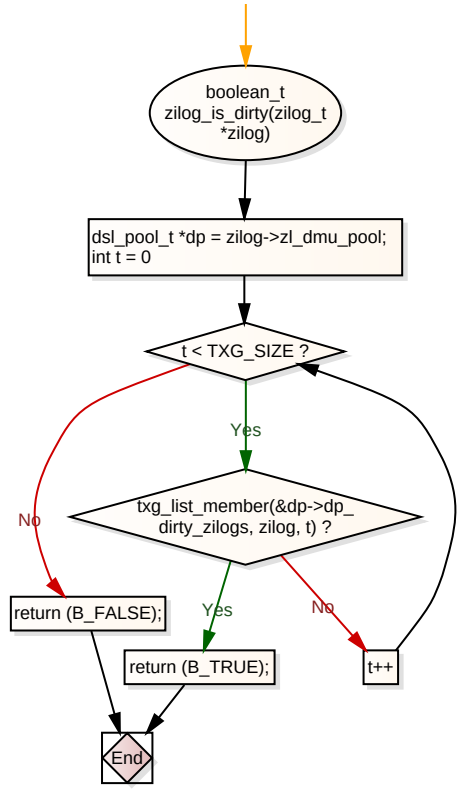

static int zil_claim_log_record (zilog_t *zilog, lr_t *lrc, void *tx, uint64_t first_txg)
lr_write_t *lr = (lr_write_t *)lrc;
int error;
lrc->lrc_txtype != TX_WRITE ?
No
Yes
* If the block is not readable, don't claim it. This can happen
* in normal operation when a log block is written to disk before
* some of the dmu_sync() blocks it points to. In this case, the
* transaction cannot have been committed to anyone (we would have
* waited for all writes to be stable first), so it is semantically
* correct to declare this the end of the log.
lr->lr_blkptr.blk_birth >= first_txg && (error = zil_read_log_data(zilog, lr, NULL)) != 0 ?
No
Yes
return (0);
return (zil_claim_log_block(zilog, &lr->lr_blkptr, tx, first_txg));
return (error);
End

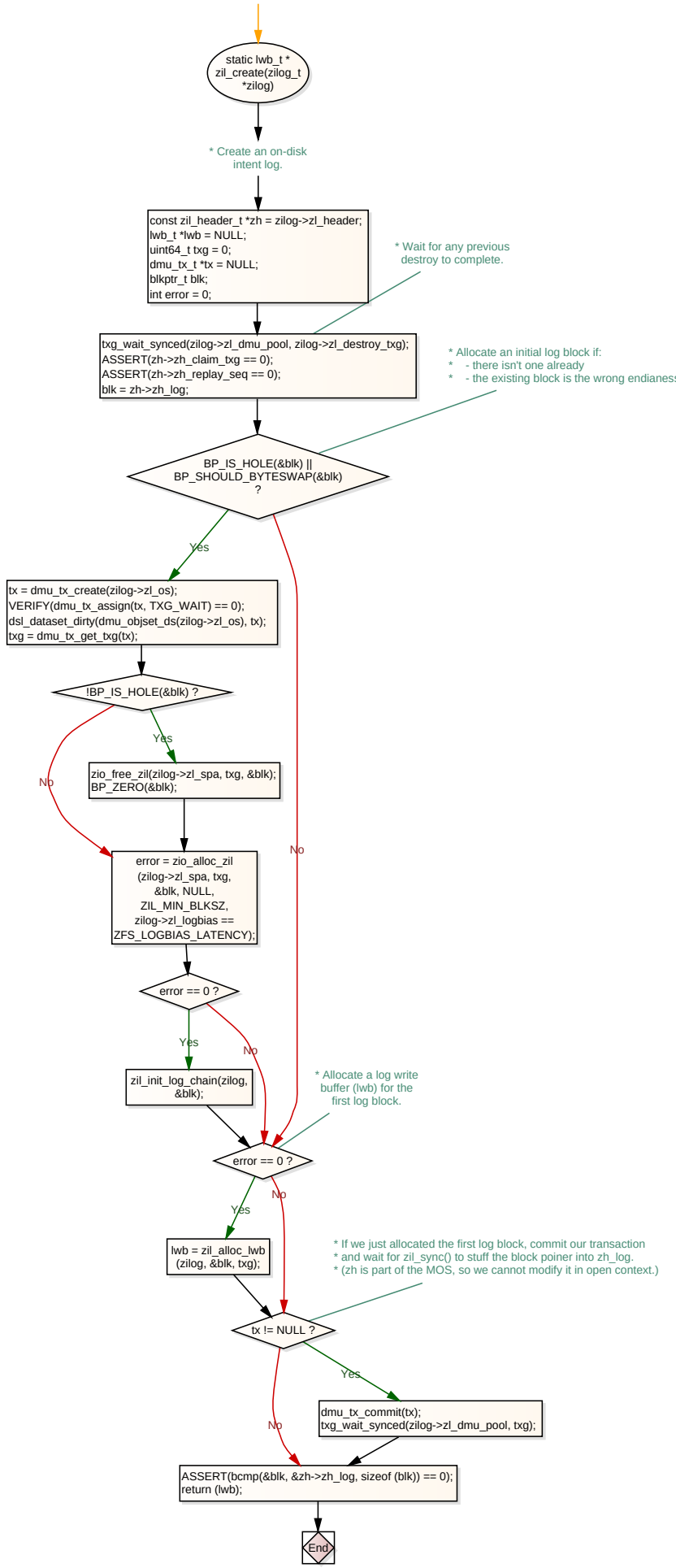

static int
zil_free_log_block(zilog_
t *zilog, blkptr_t *bp,
void *tx, uint64_t
claim_txg)
ARGSUSED
zio_free_zil(zilog->zl_spa, dmu_tx_get_txg(tx), bp);
return (0);
End

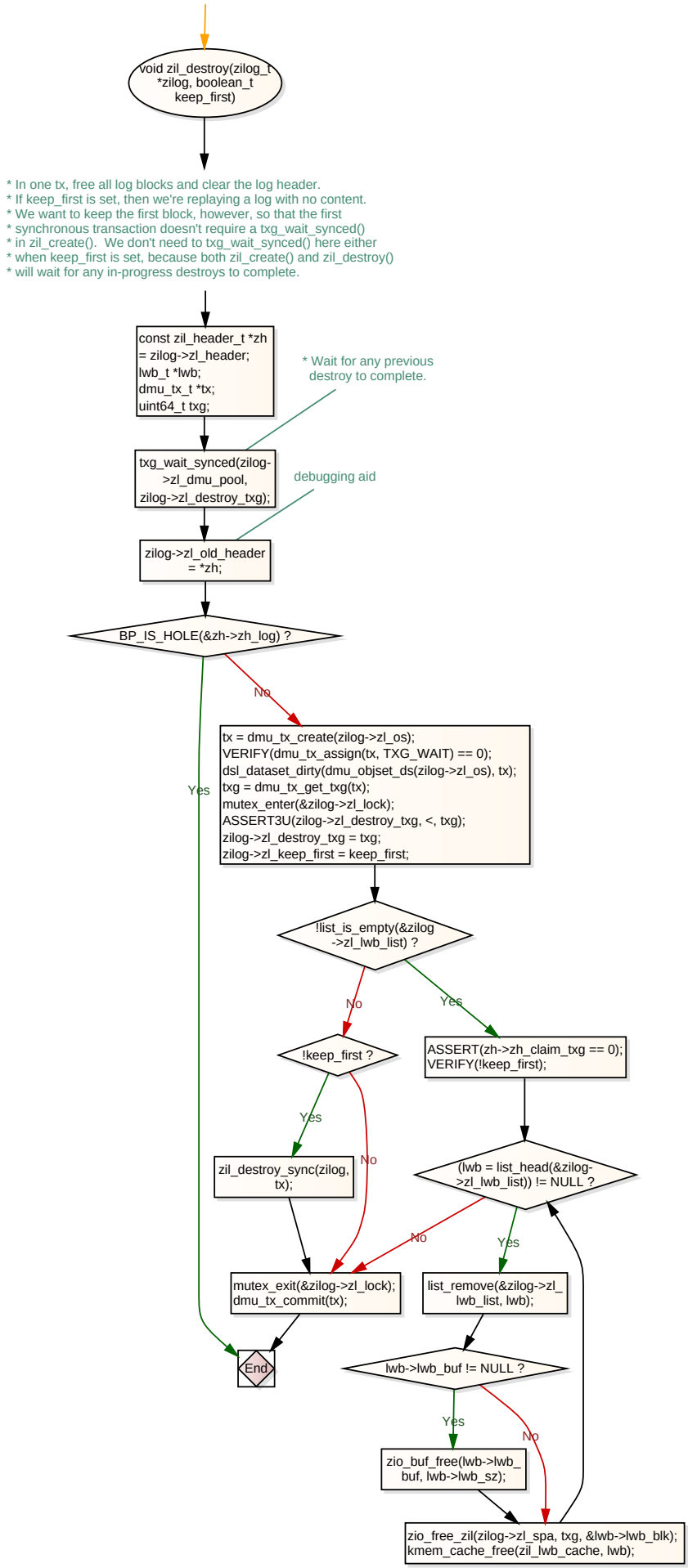

static int
zil_free_log_record
(zilog_t *zilog, lr_t
*lrc, void *tx,
uint64_t claim_txg)
lr_write_t *lr = (lr_write_t *)lrc;
blkptr_t *bp = &lr->lr_blkptr;
* If we previously
claimed it, we
need to free it.
claim_txg != 0 &&
lrc->lrc_txtype ==
TX_WRITE &&
bp->blk_birth >=
claim_txg &&
zil_bp_tree_add(zilog,
bp) == 0 ?
Yes
No
zio_free(zilog->zl_spa,
dmu_tx_get_txg(tx), bp);
return (0);
End

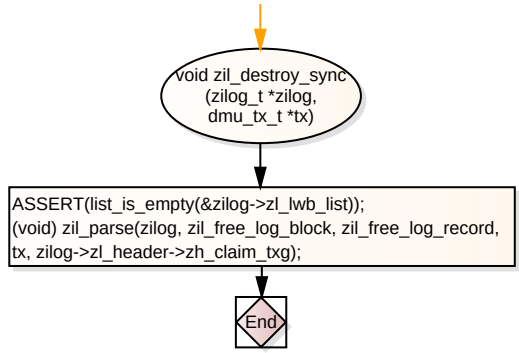

static lwb_t *
zil_alloc_lwb(zilog_t
*zilog, blkptr_t
*bp, uint64_t txg)
lwb_t *lwb;
lwb = kmem_cache_alloc(zil_lwb_cache, KM_SLEEP);
lwb->lwb_zilog = zilog;
lwb->lwb_blk = *bp;
lwb->lwb_buf = zio_buf_alloc(BP_GET_LSIZE(bp));
lwb->lwb_max_txg = txg;
lwb->lwb_zio = NULL;
lwb->lwb_tx = NULL;
BP_GET_CHECKSUM(bp) ==
ZIO_CHECKSUM_ZILOG2 ?
Yes
No
lwb->lwb_nused = sizeof (zil_chain_t);
lwb->lwb_sz = BP_GET_LSIZE(bp);
lwb->lwb_nused = 0;
lwb->lwb_sz = BP_GET_LSIZE(bp) - sizeof (zil_chain_t);
mutex_enter(&zilog->zl_lock);
list_insert_tail(&zilog->zl_lwb_list, lwb);
mutex_exit(&zilog->zl_lock);
return (lwb);
End

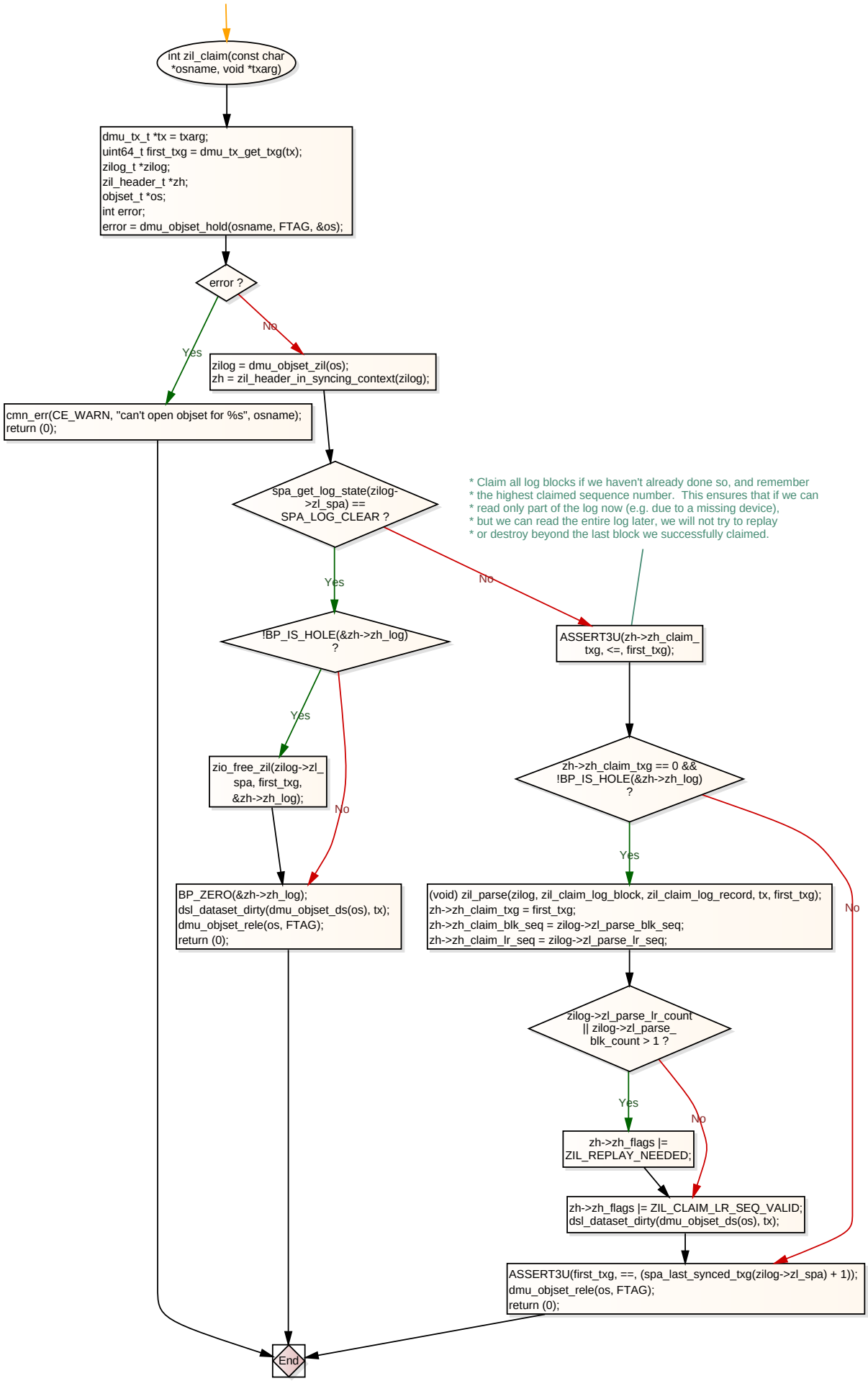

void zilog_dirty(zilog_t *zilog, uint64_t txg)
* Called when we create in-memory log transactions so that we know * to cleanup the itxs at the end of spa_sync().
dsl_pool_t *dp = zilog->zl_dmu_pool; dsl_dataset_t *ds = dmu_objset_ds(zilog->zl_os);
dsl_dataset_ is_snapshot(ds) ?
Yes
No
panic("dirtying snapshot!");
txg_list_add(&dp->dp_ dirty_zilogs, zilog, txg) == 0 ?
up the hold count until we can be written out
Yes
No
dmu_buf_add_ref(ds->ds_dbuf, zilog);
End

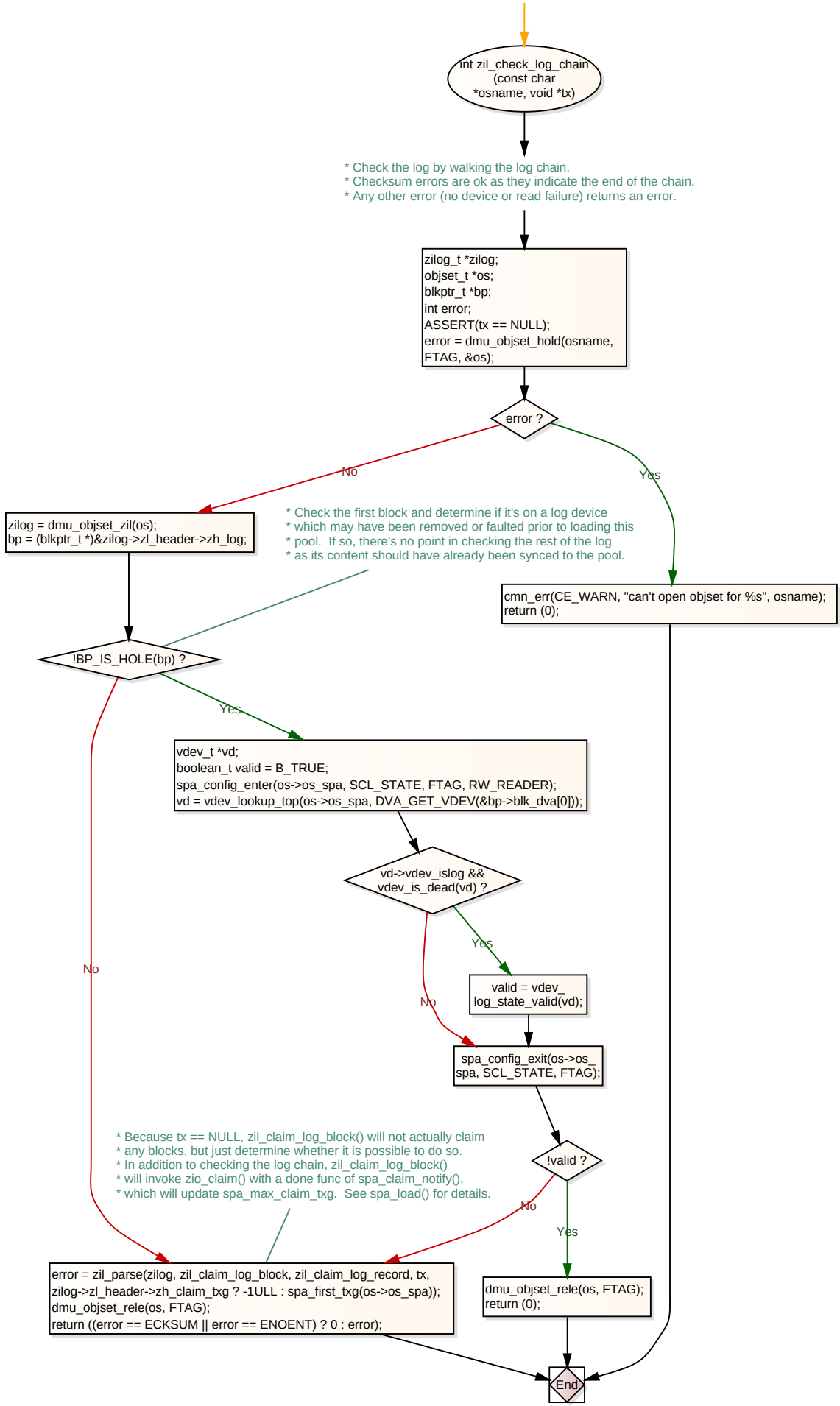

boolean_t
zilog_is_dirty(zilog_t
*zilog)
dsl_pool_t *dp = zilog->zl_dmu_pool;
int t = 0
t < TXG_SIZE ?
Yes
No
txg_list_member(&dp->dp_
dirty_zilogs, zilog, t) ?
return (B_FALSE);
Yes
No
return (B_TRUE);
t++
End

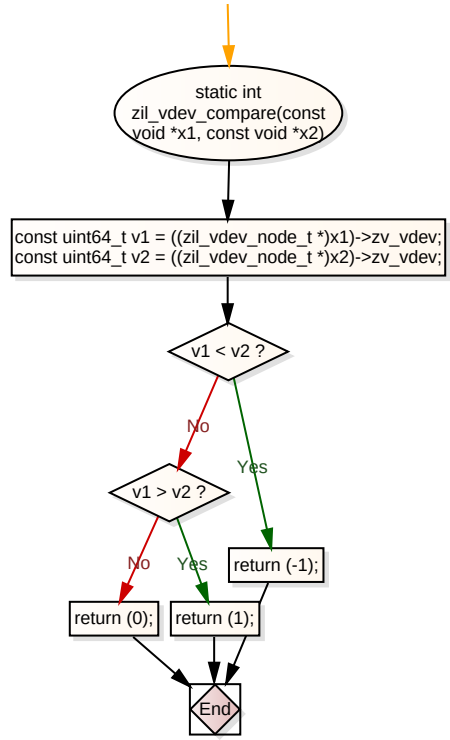

static lwb_t *
zil_create(zilog_t
*zilog)
* Create an on-disk
intent log.
const zil_header_t *zh = zilog->zl_header;
lwb_t *lwb = NULL;
uint64_t txg = 0;
dmu_tx_t *tx = NULL;
blkptr_t blk;
int error = 0;
* Wait for any previous
destroy to complete.
txg_wait_synced(zilog->zl_dmu_pool, zilog->zl_destroy_txg);
ASSERT(zh->zh_claim_txg == 0);
ASSERT(zh->zh_replay_seq == 0);
blk = zh->zh_log;
* Allocate an initial log block if:
* - there isn't one already
* - the existing block is the wrong endianess
BP_IS_HOLE(&blk) ||
BP_SHOULD_BYTESWAP(&blk)
?
Yes
No
tx = dmu_tx_create(zilog->zl_os);
VERIFY(dmu_tx_assign(tx, TXG_WAIT) == 0);
dsl_dataset_dirty(dmu_objset_ds(zilog->zl_os), tx);
txg = dmu_tx_get_txg(tx);
!BP_IS_HOLE(&blk) ?
Yes
No
zio_free_zil(zilog->zl_spa, txg, &blk);
BP_ZERO(&blk);
error = zio_alloc_zil
(zilog->zl_spa, txg,
&blk, NULL,
ZIL_MIN_BLKSZ,
zilog->zl_logbias ==
ZFS_LOGBIAS_LATENCY);
error == 0 ?
Yes
No
zil_init_log_chain(zilog,
&blk);
* Allocate a log write
buffer (lwb) for the
first log block.
error == 0 ?
Yes
No
lwb = zil_alloc_lwb
(zilog, &blk, txg);
* If we just allocated the first log block, commit our transaction
* and wait for zil_sync() to stuff the block poiner into zh_log.
* (zh is part of the MOS, so we cannot modify it in open context.)
tx != NULL ?
Yes
No
dmu_tx_commit(tx);
txg_wait_synced(zilog->zl_dmu_pool, txg);
ASSERT(bcmp(&blk, &zh->zh_log, sizeof (blk)) == 0);
return (lwb);
End

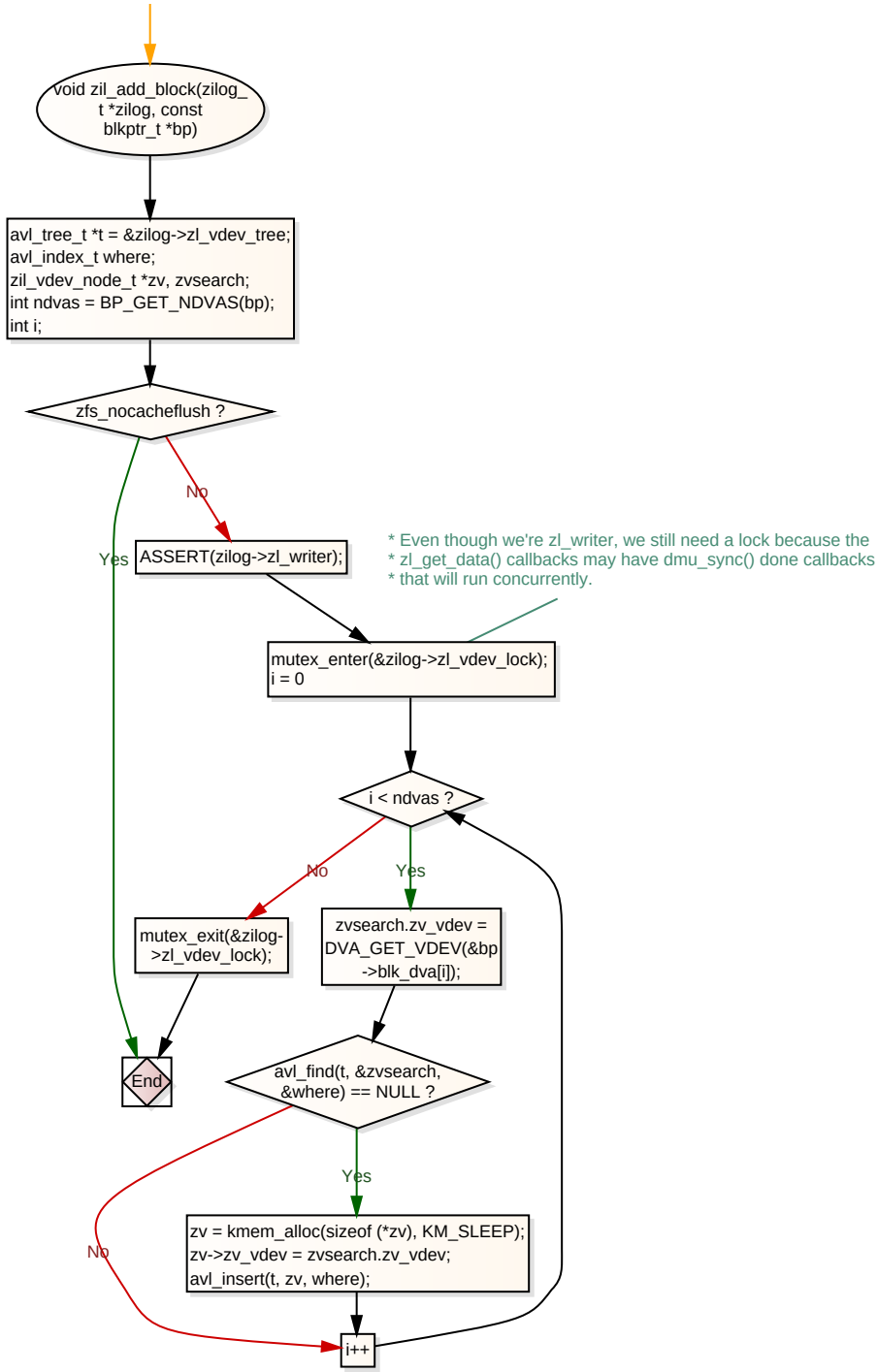

void zil_destroy(zilog_t *zilog, boolean_t keep_first)
* In one tx, free all log blocks and clear the log header.
* If keep_first is set, then we're replaying a log with no content.
* We want to keep the first block, however, so that the first
* synchronous transaction doesn't require a txg_wait_synced()
* in zil_create(). We don't need to txg_wait_synced() here either
* when keep_first is set, because both zil_create() and zil_destroy()
* will wait for any in-progress destroys to complete.
const zil_header_t *zh = zilog->zl_header;
lwb_t *lwb;
dmu_tx_t *tx;
uint64_t txg;
* Wait for any previous destroy to complete.
txg_wait_synced(zilog->zl_dmu_pool, zilog->zl_destroy_txg);
debugging aid
zilog->zl_old_header = *zh;
BP_IS_HOLE(&zh->zh_log) ?
No
Yes
tx = dmu_tx_create(zilog->zl_os);
VERIFY(dmu_tx_assign(tx, TXG_WAIT) == 0);
dsl_dataset_dirty(dmu_objset_ds(zilog->zl_os), tx);
txg = dmu_tx_get_txg(tx);
mutex_enter(&zilog->zl_lock);
ASSERT3U(zilog->zl_destroy_txg, <, txg);
zilog->zl_destroy_txg = txg;
zilog->zl_keep_first = keep_first;
!list_is_empty(&zilog->zl_lwb_list) ?
No
Yes
!keep_first ?
ASSERT(zh->zh_claim_txg == 0);
VERIFY(!keep_first);
Yes
No
zil_destroy_sync(zilog, tx);
(lwb = list_head(&zilog->zl_lwb_list)) != NULL ?
No
Yes
mutex_exit(&zilog->zl_lock);
dmu_tx_commit(tx);
list_remove(&zilog->zl_lwb_list, lwb);
End
lwb->lwb_buf != NULL ?
Yes
No
zio_buf_free(lwb->lwb_buf, lwb->lwb_sz);
zio_free_zil(zilog->zl_spa, txg, &lwb->lwb_blk);
kmem_cache_free(zil_lwb_cache, lwb);

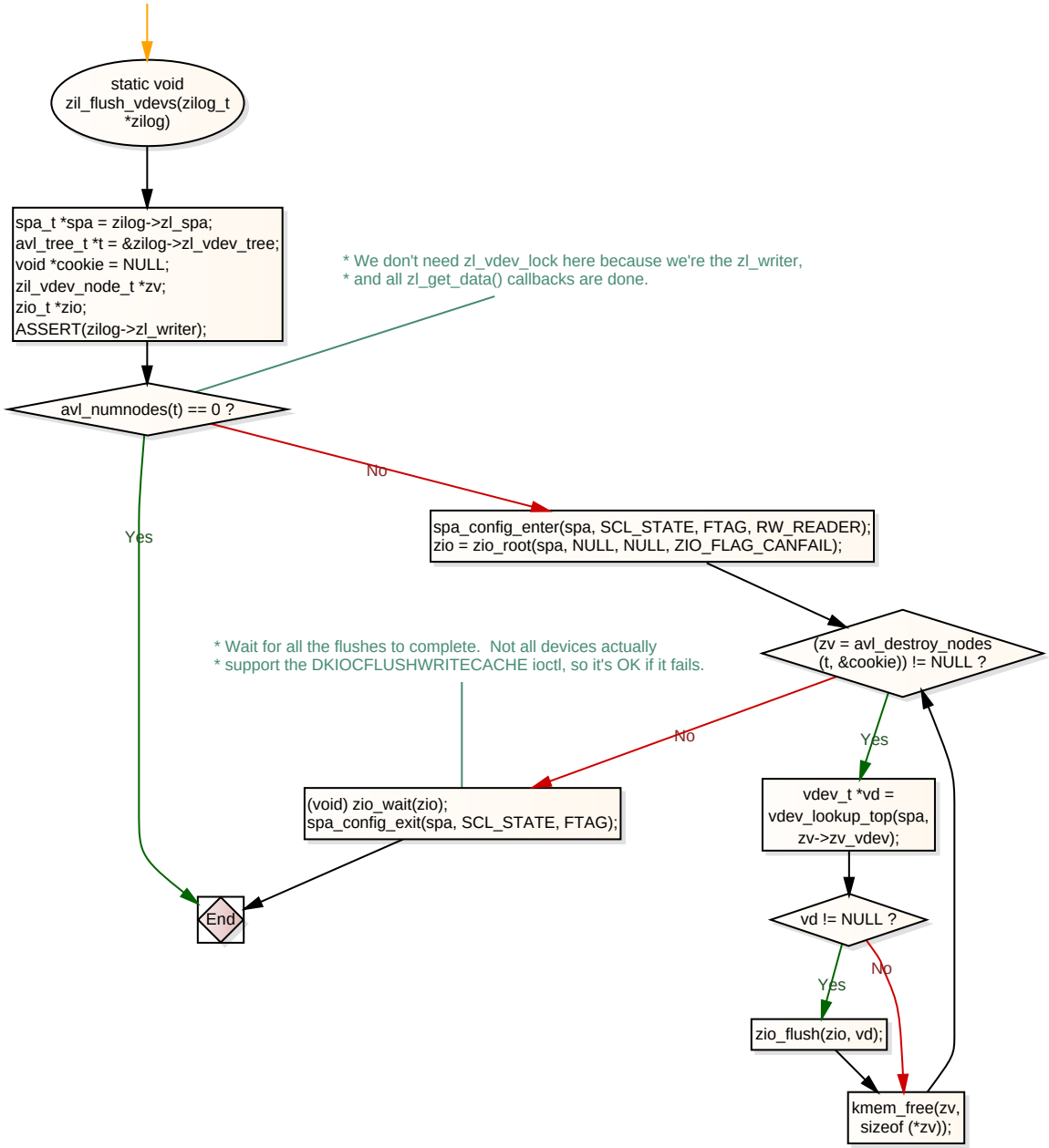

void zil_destroy_sync
(zilog_t *zilog,
dmu_tx_t *tx)
ASSERT(list_is_empty(&zilog->zl_lwb_list));
(void) zil_parse(zilog, zil_free_log_block, zil_free_log_record,
tx, zilog->zl_header->zh_claim_txg);
End

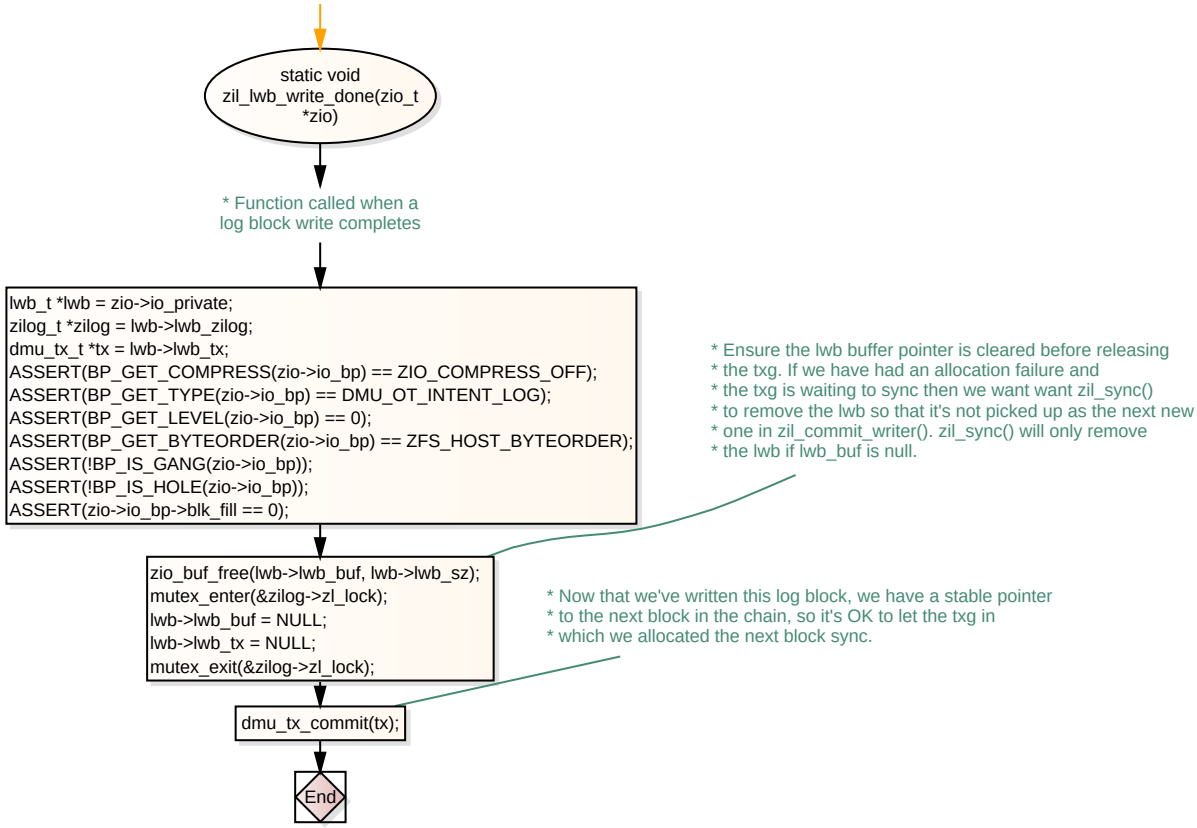

int zil_claim(const char *osname, void *txarg)
dmu_tx_t *tx = txarg;
uint64_t first_txg = dmu_tx_get_txg(tx);
zilog_t *zilog;
zil_header_t *zh;
objset_t *os;
int error;
error = dmu_objset_hold(osname, FTAG, &os);
error ?
Yes
No
cmn_err(CE_WARN, "can't open objset for %s", osname);
return (0);
zilog = dmu_objset_zil(os);
zh = zil_header_in_syncing_context(zilog);
spa_get_log_state(zilog->zl_spa) == SPA_LOG_CLEAR ?
Yes
No
* Claim all log blocks if we haven't already done so, and remember
* the highest claimed sequence number. This ensures that if we can
* read only part of the log now (e.g. due to a missing device),
* but we can read the entire log later, we will not try to replay
* or destroy beyond the last block we successfully claimed.
!BP_IS_HOLE(&zh->zh_log) ?
Yes
No
zio_free_zil(zilog->zl_spa, first_txg, &zh->zh_log);
BP_ZERO(&zh->zh_log);
dsl_dataset_dirty(dmu_objset_ds(os), tx);
dmu_objset_rele(os, FTAG);
return (0);
ASSERT3U(zh->zh_claim_txg, <=, first_txg);
zh->zh_claim_txg == 0 && !BP_IS_HOLE(&zh->zh_log) ?
Yes
No
(void) zil_parse(zilog, zil_claim_log_block, zil_claim_log_record, tx, first_txg);
zh->zh_claim_txg = first_txg;
zh->zh_claim_blk_seq = zilog->zl_parse_blk_seq;
zh->zh_claim_lr_seq = zilog->zl_parse_lr_seq;
zilog->zl_parse_lr_count || zilog->zl_parse_blk_count > 1 ?
Yes
No
zh->zh_flags |= ZIL_REPLAY_NEEDED;
zh->zh_flags |= ZIL_CLAIM_LR_SEQ_VALID;
dsl_dataset_dirty(dmu_objset_ds(os), tx);
ASSERT3U(first_txg, ==, (spa_last_synced_txg(zilog->zl_spa) + 1));
dmu_objset_rele(os, FTAG);
return (0);
End

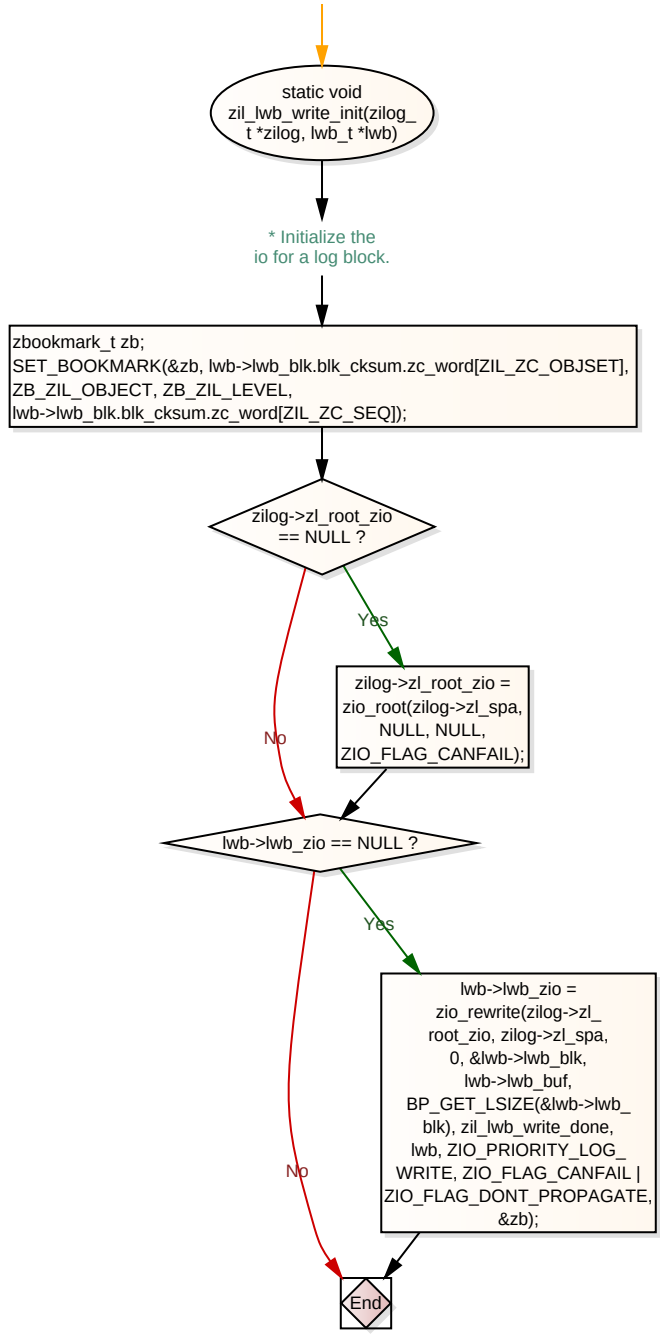

int zil_check_log_chain (const char *osname, void *tx)
* Check the log by walking the log chain.
* Checksum errors are ok as they indicate the end of the chain.
* Any other error (no device or read failure) returns an error.
zilog_t *zilog;
objset_t *os;
blkptr_t *bp;
int error;
ASSERT(tx == NULL);
error = dmu_objset_hold(osname, FTAG, &os);
error ?
No
Yes
zilog = dmu_objset_zil(os);
bp = (blkptr_t *)&zilog->zl_header->zh_log;
* Check the first block and determine if it's on a log device
* which may have been removed or faulted prior to loading this
* pool. If so, there's no point in checking the rest of the log
* as its content should have already been synced to the pool.
cmn_err(CE_WARN, "can't open objset for %s", osname);
return (0);
!BP_IS_HOLE(bp) ?
Yes
No
vdev_t *vd;
boolean_t valid = B_TRUE;
spa_config_enter(os->os_spa, SCL_STATE, FTAG, RW_READER);
vd = vdev_lookup_top(os->os_spa, DVA_GET_VDEV(&bp->blk_dva[0]));
vd->vdev_islog && vdev_is_dead(vd) ?
Yes
No
valid = vdev_ log_state_valid(vd);
spa_config_exit(os->os_ spa, SCL_STATE, FTAG);
!valid ?
No
Yes
* Because tx == NULL, zil_claim_log_block() will not actually claim
* any blocks, but just determine whether it is possible to do so.
* In addition to checking the log chain, zil_claim_log_block()
* will invoke zio_claim() with a done func of spa_claim_notify(),
* which will update spa_max_claim_txg. See spa_load() for details.
error = zil_parse(zilog, zil_claim_log_block, zil_claim_log_record, tx, zilog->zl_header->zh_claim_txg ? -1ULL : spa_first_txg(os->os_spa));
dmu_objset_rele(os, FTAG);
return ((error == ECKSUM || error == ENOENT) ? 0 : error);
dmu_objset_rele(os, FTAG);
return (0);
End

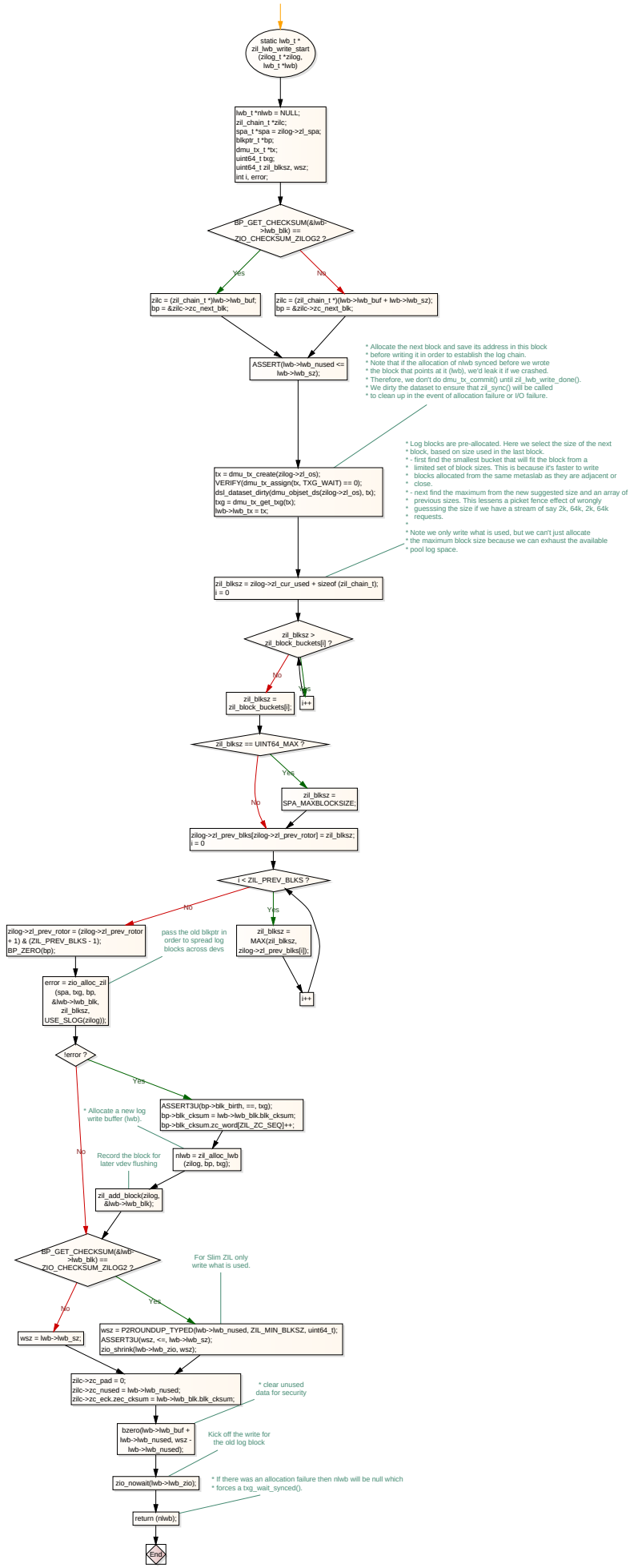

static int
zil_vdev_compare(const
void *x1, const void *x2)
const uint64_t v1 = ((zil_vdev_node_t *)x1)->zv_vdev;
const uint64_t v2 = ((zil_vdev_node_t *)x2)->zv_vdev;
v1 < v2 ?
No
Yes
v1 > v2 ?
No
Yes
return (-1);
return (0);
return (1);
End

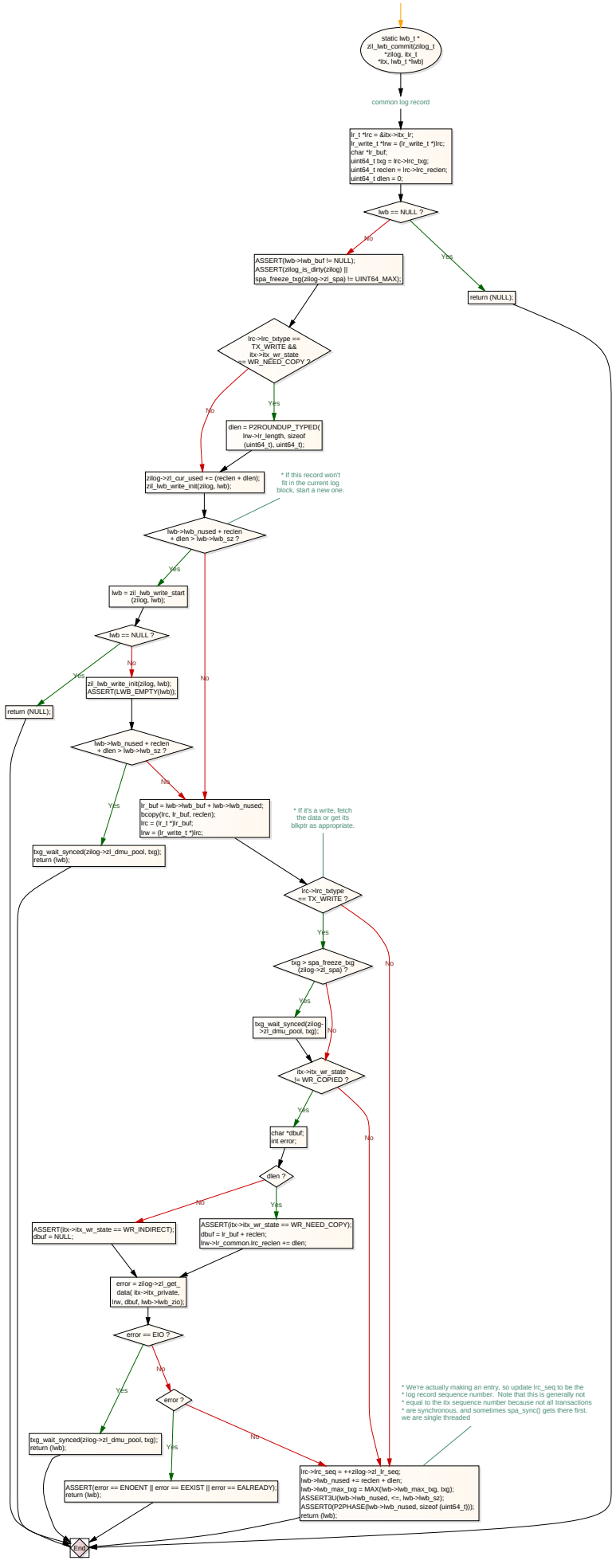

void zil_add_block(zilog_t *zilog, const blkptr_t *bp)
avl_tree_t *t = &zilog->zl_vdev_tree;
avl_index_t where;
zil_vdev_node_t *zv, zvsearch;
int ndvas = BP_GET_NDVAS(bp);
int i;
zfs_nocacheflush ?
No
Yes
ASSERT(zilog->zl_writer);
* Even though we're zl_writer, we still need a lock because the
* zl_get_data() callbacks may have dmu_sync() done callbacks
* that will run concurrently.
mutex_enter(&zilog->zl_vdev_lock);
i = 0
i < ndvas ?
No
Yes
mutex_exit(&zilog->zl_vdev_lock);
zvsearch.zv_vdev = DVA_GET_VDEV(&bp->blk_dva[i]);
End
avl_find(t, &zvsearch, &where) == NULL ?
Yes
No
zv = kmem_alloc(sizeof (*zv), KM_SLEEP);
zv->zv_vdev = zvsearch.zv_vdev;
avl_insert(t, zv, where);
i++

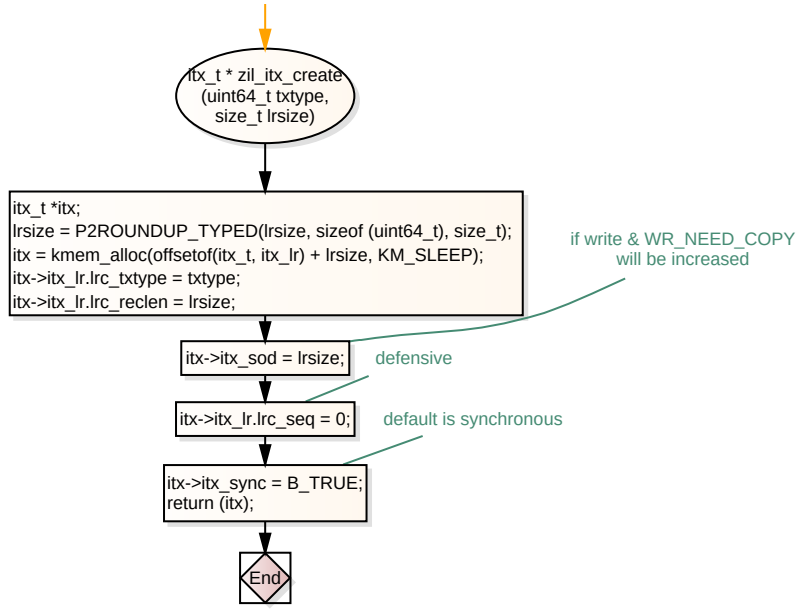

static void
zil_flush_vdevs(zilog_t
*zilog)
spa_t *spa = zilog->zl_spa;
avl_tree_t *t = &zilog->zl_vdev_tree;
void *cookie = NULL;
zil_vdev_node_t *zv;
zio_t *zio;
ASSERT(zilog->zl_writer);
* We don't need zl_vdev_lock here because we're the zl_writer,
* and all zl_get_data() callbacks are done.
avl_numnodes(t) == 0 ?
Yes
No
spa_config_enter(spa, SCL_STATE, FTAG, RW_READER);
zio = zio_root(spa, NULL, NULL, ZIO_FLAG_CANFAIL);
(zv = avl_destroy_nodes
(t, &cookie)) != NULL ?
No
Yes
* Wait for all the flushes to complete. Not all devices actually
* support the DKIOCFLUSHWRITECACHE ioctl, so it's OK if it fails.
(void) zio_wait(zio);
spa_config_exit(spa, SCL_STATE, FTAG);
vdev_t *vd =
vdev_lookup_top(spa,
zv->zv_vdev);
vd != NULL ?
Yes
No
zio_flush(zio, vd);
kmem_free(zv,
sizeof (*zv));
End

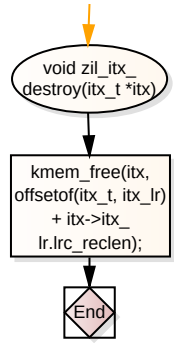

static void
zil_lwb_write_done(zio_t
*zio)
* Function called when a
log block write completes
lwb_t *lwb = zio->io_private;
zilog_t *zilog = lwb->lwb_zilog;
dmu_tx_t *tx = lwb->lwb_tx;
ASSERT(BP_GET_COMPRESS(zio->io_bp) == ZIO_COMPRESS_OFF);
ASSERT(BP_GET_TYPE(zio->io_bp) == DMU_OT_INTENT_LOG);
ASSERT(BP_GET_LEVEL(zio->io_bp) == 0);
ASSERT(BP_GET_BYTEORDER(zio->io_bp) == ZFS_HOST_BYTEORDER);
ASSERT(!BP_IS_GANG(zio->io_bp));
ASSERT(!BP_IS_HOLE(zio->io_bp));
ASSERT(zio->io_bp->blk_fill == 0);
* Ensure the lwb buffer pointer is cleared before releasing
* the txg. If we have had an allocation failure and
* the txg is waiting to sync then we want want zil_sync()
* to remove the lwb so that it's not picked up as the next new
* one in zil_commit_writer(). zil_sync() will only remove
* the lwb if lwb_buf is null.
zio_buf_free(lwb->lwb_buf, lwb->lwb_sz);
mutex_enter(&zilog->zl_lock);
lwb->lwb_buf = NULL;
lwb->lwb_tx = NULL;
mutex_exit(&zilog->zl_lock);
* Now that we've written this log block, we have a stable pointer
* to the next block in the chain, so it's OK to let the txg in
* which we allocated the next block sync.
dmu_tx_commit(tx);
End

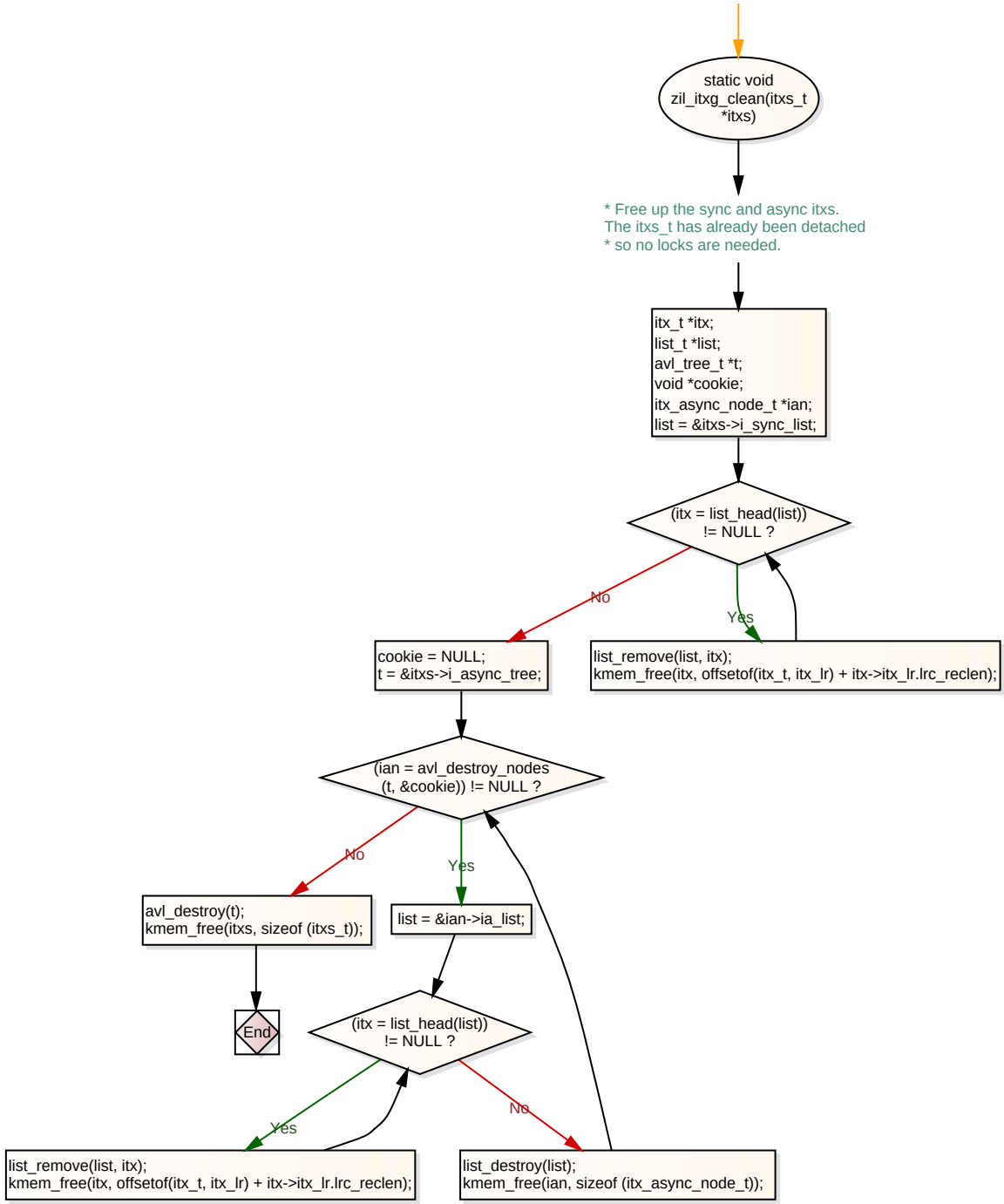

static void
zil_lwb_write_init(zilog_
t *zilog, lwb_t *lwb)
* Initialize the
io for a log block.
zbookmark_t zb;
SET_BOOKMARK(&zb, lwb->lwb_blk.blk_cksum.zc_word[ZIL_ZC_OBJSET],
ZB_ZIL_OBJECT, ZB_ZIL_LEVEL,
lwb->lwb_blk.blk_cksum.zc_word[ZIL_ZC_SEQ]);
zilog->zl_root_zio
== NULL ?
Yes
zilog->zl_root_zio =
zio_root(zilog->zl_spa,
NULL, NULL,
ZIO_FLAG_CANFAIL);
No
lwb->lwb_zio == NULL ?
Yes
lwb->lwb_zio =
zio_rewrite(zilog->zl_
root_zio, zilog->zl_spa,
0, &lwb->lwb_blk,
lwb->lwb_buf,
BP_GET_LSIZE(&lwb->lwb_
blk), zil_lwb_write_done,
lwb, ZIO_PRIORITY_LOG_
WRITE, ZIO_FLAG_CANFAIL |
ZIO_FLAG_DONT_PROPAGATE,
&zb);
No
End

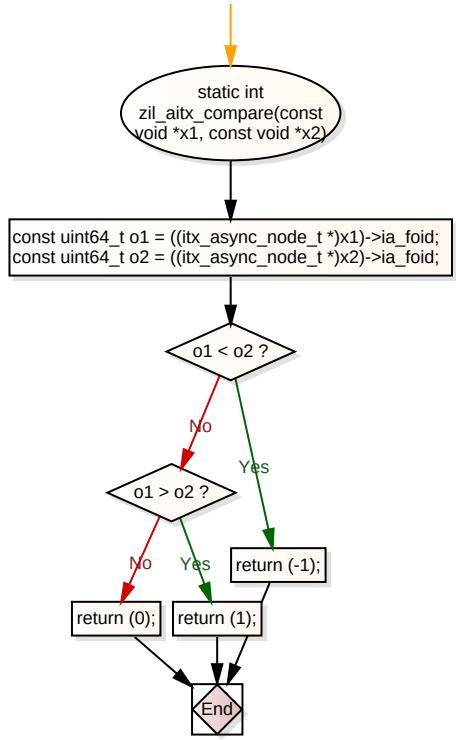

static lwb_t *
zil_lwb_write_start
(zilog_t *zilog,
lwb_t *lwb)
lwb_t *nlwb = NULL;
zil_chain_t *zilc;
spa_t *spa = zilog->zl_spa;
blkptr_t *bp;
dmu_tx_t *tx;
uint64_t txg;
uint64_t zil_blksz, wsz;
int i, error;
BP_GET_CHECKSUM(&lwb->lwb_blk) ==
ZIO_CHECKSUM_ZILOG2 ?
Yes
No
zilc = (zil_chain_t *)lwb->lwb_buf;
bp = &zilc->zc_next_blk;
zilc = (zil_chain_t *)(lwb->lwb_buf + lwb->lwb_sz);
bp = &zilc->zc_next_blk;
ASSERT(lwb->lwb_nused <=
lwb->lwb_sz);
* Allocate the next block and save its address in this block
* before writing it in order to establish the log chain.
* Note that if the allocation of nlwb synced before we wrote
* the block that points at it (lwb), we'd leak it if we crashed.
* Therefore, we don't do dmu_tx_commit() until zil_lwb_write_done().
* We dirty the dataset to ensure that zil_sync() will be called
* to clean up in the event of allocation failure or I/O failure.
tx = dmu_tx_create(zilog->zl_os);
VERIFY(dmu_tx_assign(tx, TXG_WAIT) == 0);
dsl_dataset_dirty(dmu_objset_ds(zilog->zl_os), tx);
txg = dmu_tx_get_txg(tx);
lwb->lwb_tx = tx;
* Log blocks are pre-allocated. Here we select the size of the next
* block, based on size used in the last block.
* - first find the smallest bucket that will fit the block from a
* limited set of block sizes. This is because it's faster to write
* blocks allocated from the same metaslab as they are adjacent or
* close.
* - next find the maximum from the new suggested size and an array of
* previous sizes. This lessens a picket fence effect of wrongly
* guesssing the size if we have a stream of say 2k, 64k, 2k, 64k
* requests.
*
* Note we only write what is used, but we can't just allocate
* the maximum block size because we can exhaust the available
* pool log space.
zil_blksz = zilog->zl_cur_used + sizeof (zil_chain_t);
i = 0
zil_blksz >
zil_block_buckets[i] ?
No
Yes
zil_blksz =
zil_block_buckets[i];
i++
zil_blksz == UINT64_MAX ?
Yes
No
zil_blksz =
SPA_MAXBLOCKSIZE;
zilog->zl_prev_blks[zilog->zl_prev_rotor] = zil_blksz;
i = 0
i < ZIL_PREV_BLKS ?
No
Yes
zilog->zl_prev_rotor = (zilog->zl_prev_rotor
+ 1) & (ZIL_PREV_BLKS - 1);
BP_ZERO(bp);
pass the old blkptr in
order to spread log
blocks across devs
zil_blksz =
MAX(zil_blksz,
zilog->zl_prev_blks[i]);
i++
error = zio_alloc_zil
(spa, txg, bp,
&lwb->lwb_blk,
zil_blksz,
USE_SLOG(zilog));
!error ?
Yes
No
* Allocate a new log
write buffer (lwb).
ASSERT3U(bp->blk_birth, ==, txg);
bp->blk_cksum = lwb->lwb_blk.blk_cksum;
bp->blk_cksum.zc_word[ZIL_ZC_SEQ]++;
Record the block for
later vdev flushing
nlwb = zil_alloc_lwb
(zilog, bp, txg);
zil_add_block(zilog,
&lwb->lwb_blk);
BP_GET_CHECKSUM(&lwb->lwb_blk) ==
ZIO_CHECKSUM_ZILOG2 ?
For Slim ZIL only
write what is used.
No
Yes
wsz = lwb->lwb_sz;
wsz = P2ROUNDUP_TYPED(lwb->lwb_nused, ZIL_MIN_BLKSZ, uint64_t);
ASSERT3U(wsz, <=, lwb->lwb_sz);
zio_shrink(lwb->lwb_zio, wsz);
zilc->zc_pad = 0;
zilc->zc_nused = lwb->lwb_nused;
zilc->zc_eck.zec_cksum = lwb->lwb_blk.blk_cksum;
* clear unused
data for security
bzero(lwb->lwb_buf +
lwb->lwb_nused, wsz -
lwb->lwb_nused);
Kick off the write for
the old log block
zio_nowait(lwb->lwb_zio);
* If there was an allocation failure then nlwb will be null which
* forces a txg_wait_synced().
return (nlwb);
End

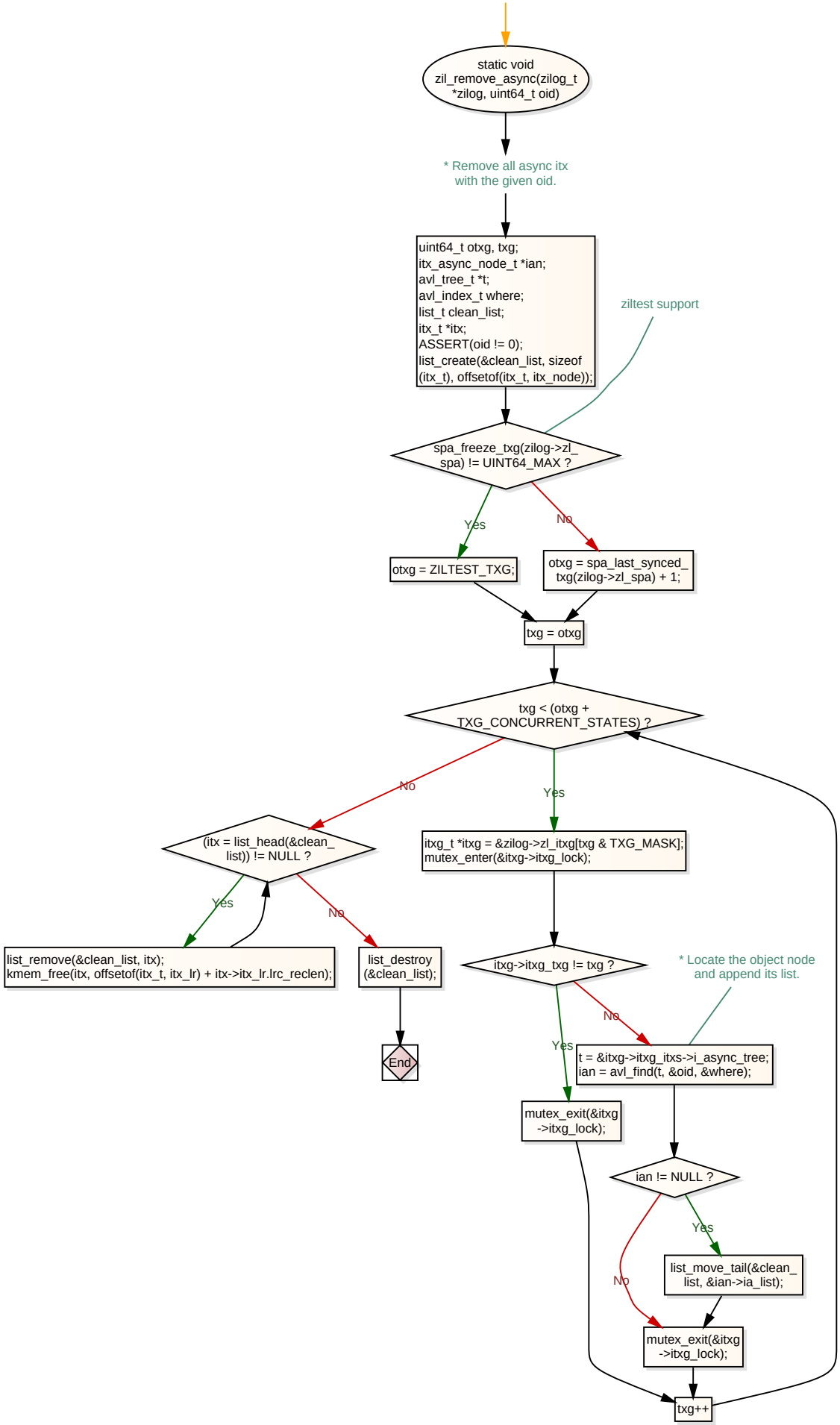

static lwb_t *
zil_lwb_commit(zilog_t
*zilog, itx_t
*itx, lwb_t *lwb)
common log record
lr_t *lrc = &itx->itx_lr;
lr_write_t *lrw = (lr_write_t *)lrc;
char *lr_buf;
uint64_t txg = lrc->lrc_txg;
uint64_t reclen = lrc->lrc_reclen;
uint64_t dlen = 0;
lwb == NULL ?
No
Yes
return (NULL);
ASSERT(lwb->lwb_buf != NULL);
ASSERT(zilog_is_dirty(zilog) ||
spa_freeze_txg(zilog->zl_spa) != UINT64_MAX);
lrc->lrc_txtype ==
TX_WRITE &&
itx->itx_wr_state
== WR_NEED_COPY ?
No
Yes
dlen = P2ROUNDUP_TYPED(
lrw->lr_length, sizeof
(uint64_t), uint64_t);
zilog->zl_cur_used += (reclen + dlen);
zil_lwb_write_init(zilog, lwb);
* If this record won't
fit in the current log
block, start a new one.
lwb->lwb_nused + reclen
+ dlen > lwb->lwb_sz ?
Yes
No
lwb = zil_lwb_write_start
(zilog, lwb);
lwb == NULL ?
Yes
No
return (NULL);
zil_lwb_write_init(zilog, lwb);
ASSERT(LWB_EMPTY(lwb));
lwb->lwb_nused + reclen
+ dlen > lwb->lwb_sz ?
Yes
No
txg_wait_synced(zilog->zl_dmu_pool, txg);
return (lwb);
lr_buf = lwb->lwb_buf + lwb->lwb_nused;
bcopy(lrc, lr_buf, reclen);
lrc = (lr_t *)lr_buf;
lrw = (lr_write_t *)lrc;
* If it's a write, fetch
the data or get its
blkptr as appropriate.
lrc->lrc_txtype
== TX_WRITE ?
Yes
No
txg > spa_freeze_txg
(zilog->zl_spa) ?
Yes
No
txg_wait_synced(zilog-
>zl_dmu_pool, txg);
itx->itx_wr_state
!= WR_COPIED ?
Yes
No
char *dbuf;
int error;
dlen ?
No
Yes
ASSERT(itx->itx_wr_state == WR_INDIRECT);
dbuf = NULL;
ASSERT(itx->itx_wr_state == WR_NEED_COPY);
dbuf = lr_buf + reclen;
lrw->lr_common.lrc_reclen += dlen;
error = zilog->zl_get_
data(itx->itx_private,
lrw, dbuf, lwb->lwb_zio);
error == EIO ?
No
Yes
error ?
Yes
No
txg_wait_synced(zilog->zl_dmu_pool, txg);
return (lwb);
ASSERT(error == ENOENT || error == EEXIST || error == EALREADY);
return (lwb);
* We're actually making an entry, so update lrc_seq to be the
* log record sequence number. Note that this is generally not
* equal to the itx sequence number because not all transactions
* are synchronous, and sometimes spa_sync() gets there first.
we are single threaded
lrc->lrc_seq = ++zilog->zl_lr_seq;
lwb->lwb_nused += reclen + dlen;
lwb->lwb_max_txg = MAX(lwb->lwb_max_txg, txg);
ASSERT3U(lwb->lwb_nused, <=, lwb->lwb_sz);
ASSERT0(P2PHASE(lwb->lwb_nused, sizeof (uint64_t)));
return (lwb);
Exit

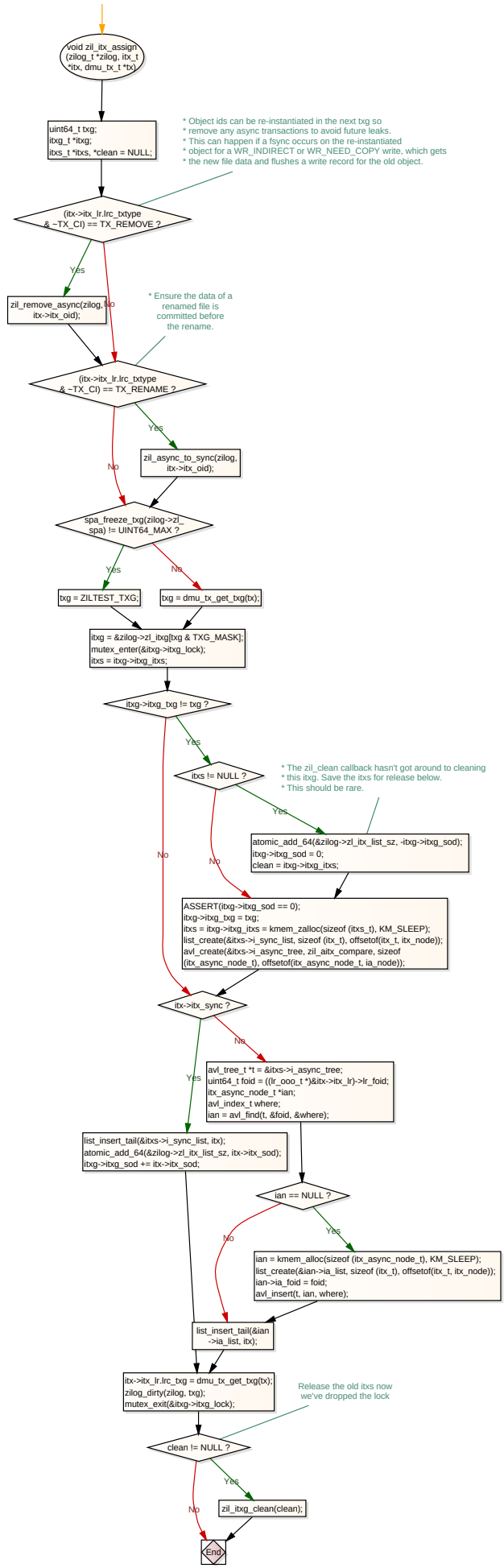

itx_t * zil_itx_create
(uint64_t txtype,
size_t lrsize)
itx_t *itx;
lrsize = P2ROUNDUP_TYPED(lrsize, sizeof (uint64_t), size_t);
itx = kmem_alloc(offsetof(itx_t, itx_lr) + lrsize, KM_SLEEP);
itx->itx_lr.lrc_txtype = txtype;
itx->itx_lr.lrc_reclen = lrsize;
if write & WR_NEED_COPY
will be increased
itx->itx_sod = lrsize;
defensive
itx->itx_lr.lrc_seq = 0;
default is synchronous
itx->itx_sync = B_TRUE;
return (itx);
End

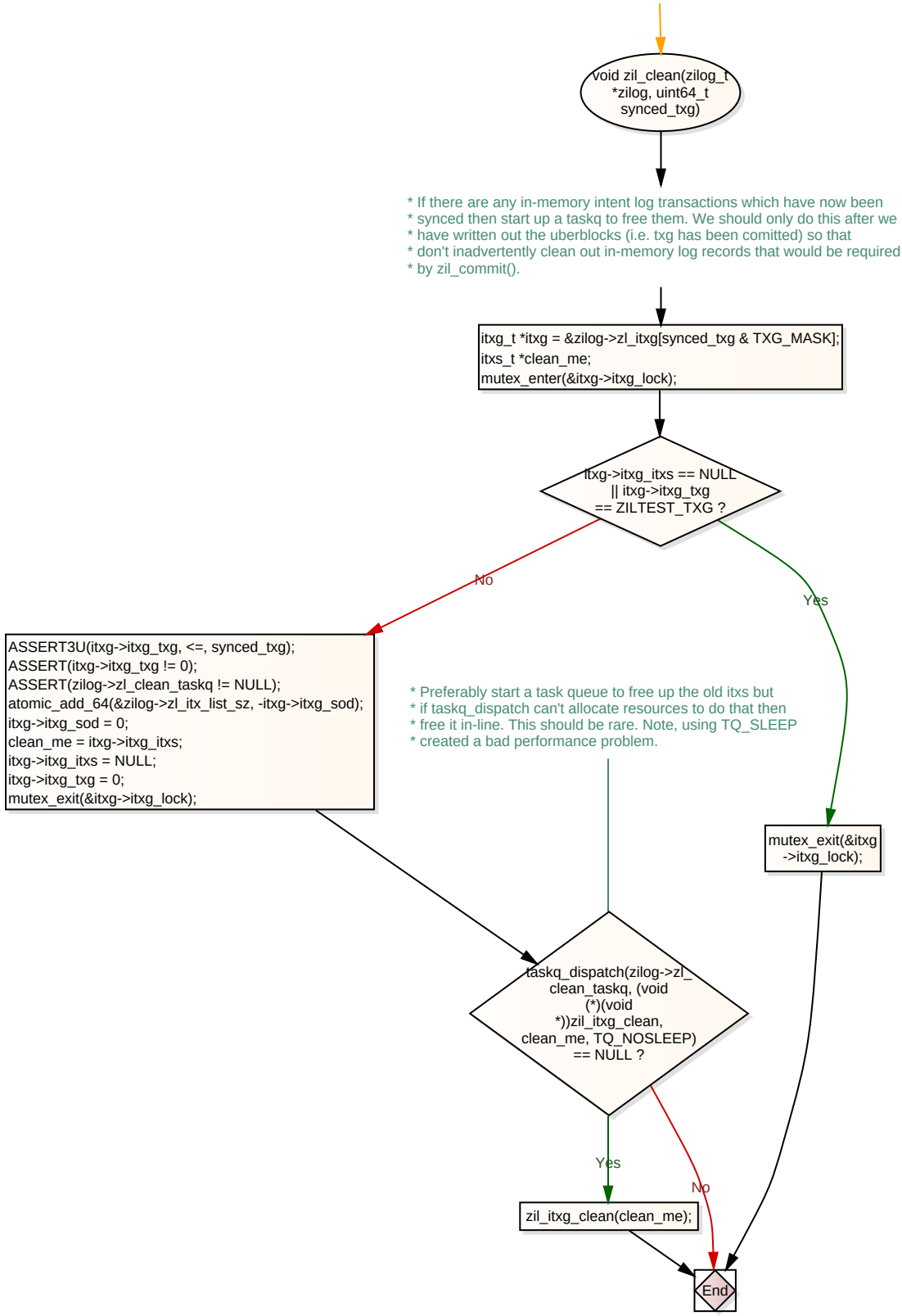

void zil_itx_
destroy(itx_t *itx)
kmem_free(itx,
offsetof(itx_t, itx_lr)
+ itx->itx_
lr.lrc_reclen);
End

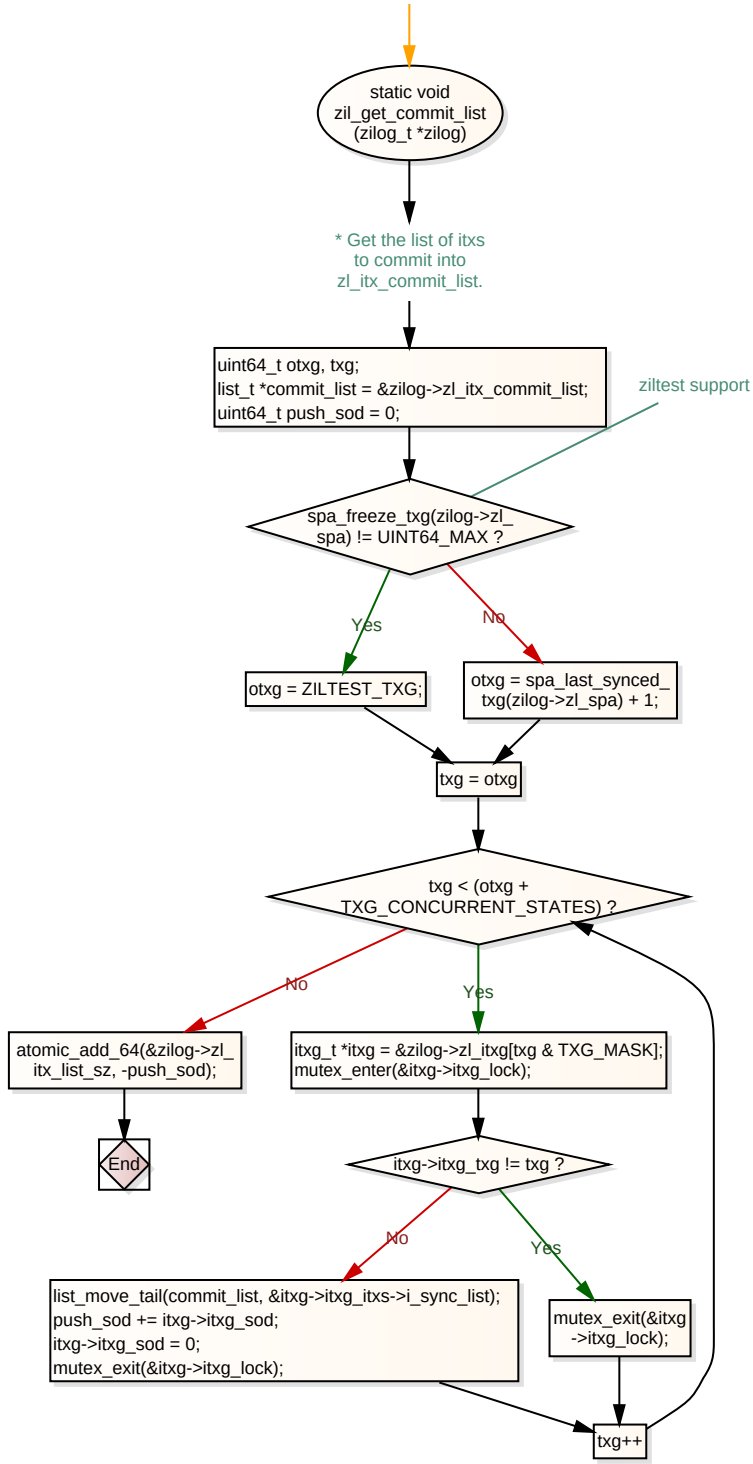

static void
zil_itxg_clean(itxs_t
*itxs)
* Free up the sync and async itxs.
The itxs_t has already been detached
* so no locks are needed.
itx_t *itx;
list_t *list;
avl_tree_t *t;
void *cookie;
itx_async_node_t *ian;
list = &itxs->i_sync_list;
(itx = list_head(list))
!= NULL ?
No
Yes
cookie = NULL;
t = &itxs->i_async_tree;
list_remove(list, itx);
kmem_free(itx, offsetof(itx_t, itx_lr) + itx->itx_lr.lrc_reclen);
(ian = avl_destroy_nodes
(t, &cookie)) != NULL ?
No
Yes
avl_destroy(t);
kmem_free(itxs, sizeof (itxs_t));
list = &ian->ia_list;
End
(itx = list_head(list))
!= NULL ?
Yes
No
list_remove(list, itx);
kmem_free(itx, offsetof(itx_t, itx_lr) + itx->itx_lr.lrc_reclen);
list_destroy(list);
kmem_free(ian, sizeof (itx_async_node_t));

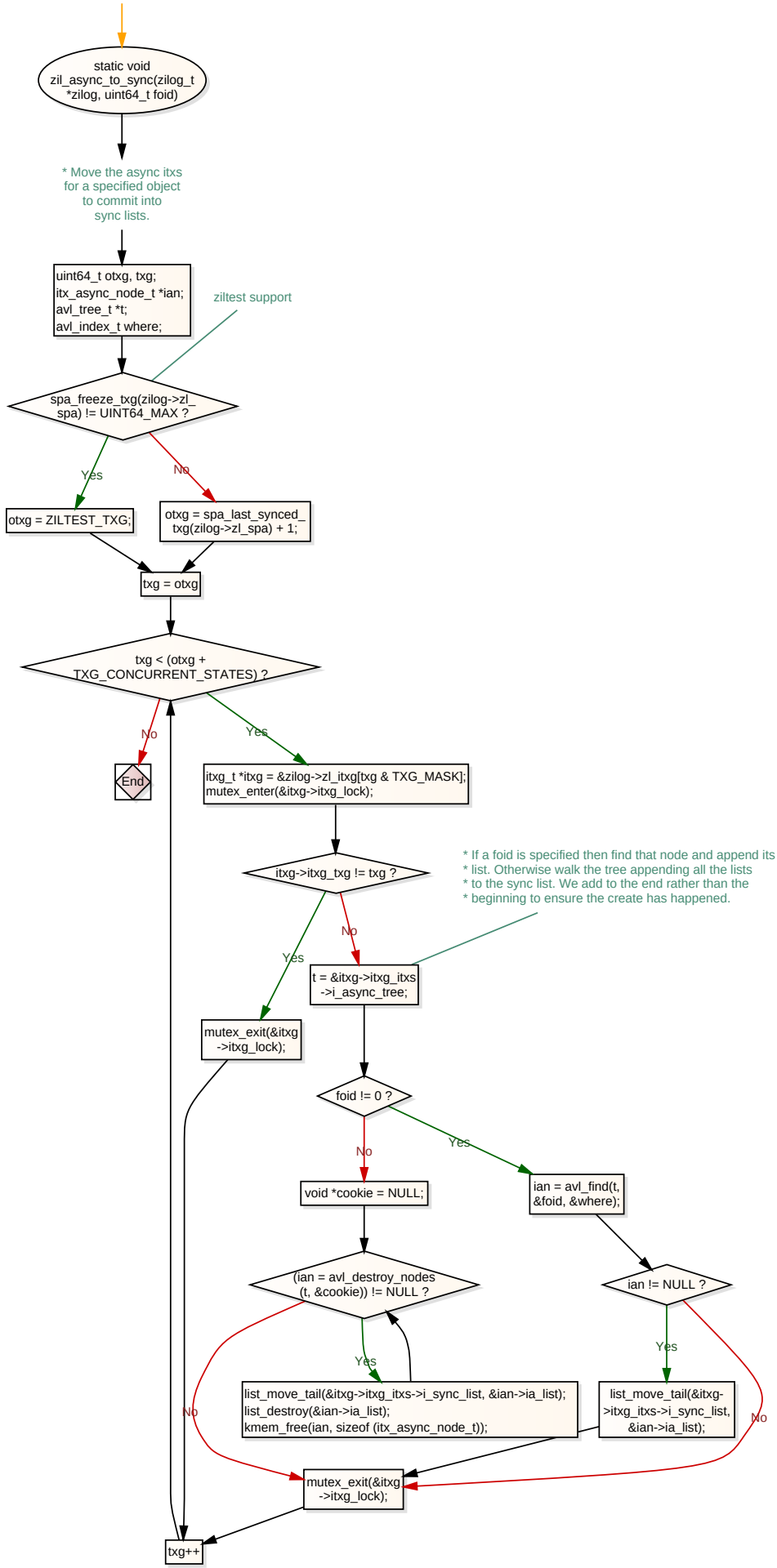

static int
zil_aitx_compare(const
void *x1, const void *x2)
const uint64_t o1 = ((itx_async_node_t *)x1)->ia_foid;
const uint64_t o2 = ((itx_async_node_t *)x2)->ia_foid;
o1 < o2 ?
No
Yes
o1 > o2 ?
No
Yes
return (-1);
return (0);
return (1);
End

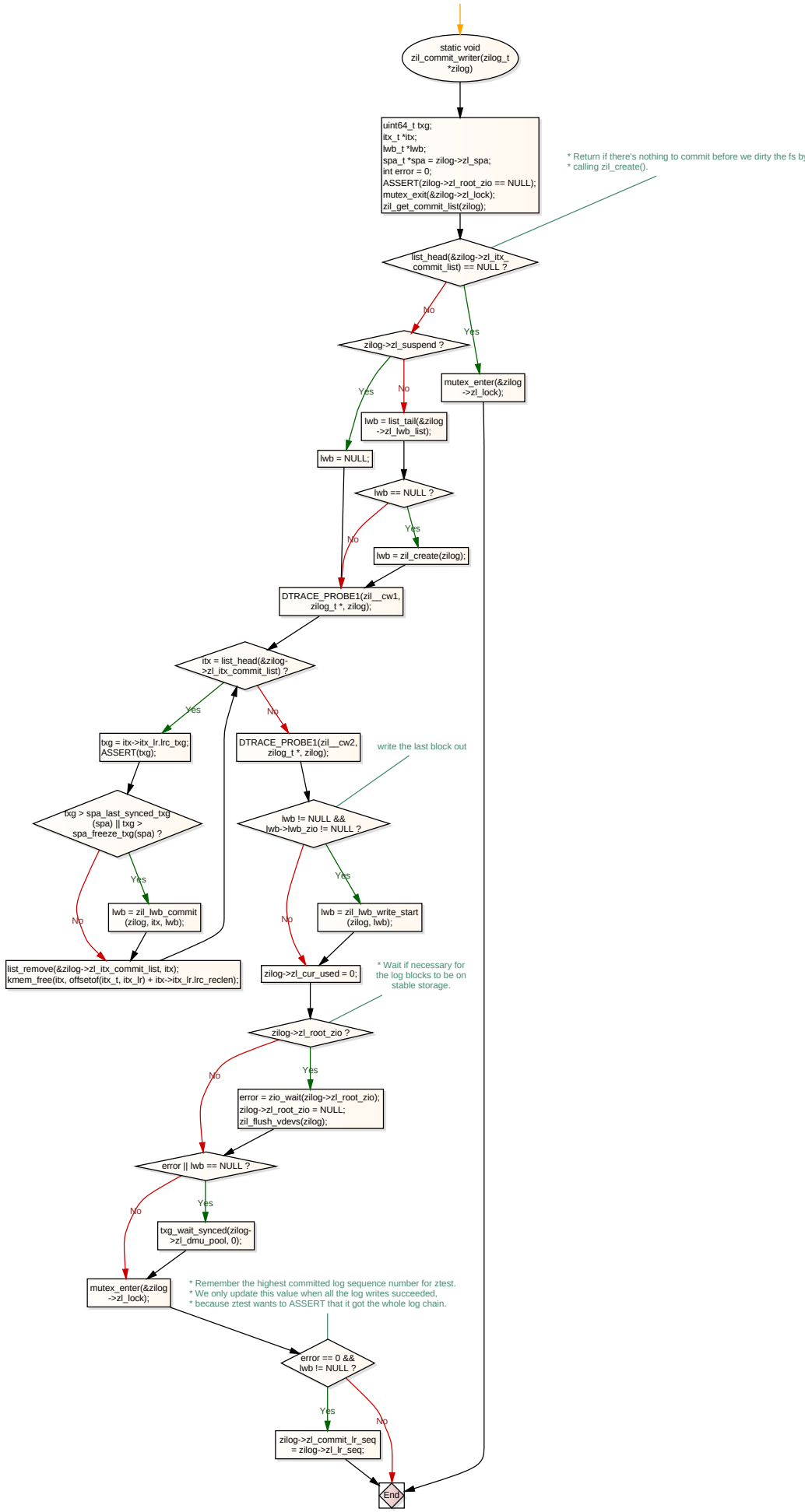

static void
zil_remove_async(zilog_t
*zilog, uint64_t oid)
* Remove all async itx
with the given oid.
uint64_t otxg, txg;
itx_async_node_t *ian;
avl_tree_t *t;
avl_index_t where;
list_t clean_list;
itx_t *itx;
ASSERT(oid != 0);
list_create(&clean_list, sizeof
(itx_t), offsetof(itx_t, itx_node));
ziltest support
spa_freeze_txg(zilog->zl_
spa) != UINT64_MAX ?
Yes
No
otxg = ZILTEST_TXG;
otxg = spa_last_synced_
txg(zilog->zl_spa) + 1;
txg = otxg
txg < (otxg +
TXG_CONCURRENT_STATES) ?
No
Yes
(itx = list_head(&clean_
list)) != NULL ?
itxg_t *itxg = &zilog->zl_itxg[txg & TXG_MASK];
mutex_enter(&itxg->itxg_lock);
Yes
No
list_remove(&clean_list, itx);
kmem_free(itx, offsetof(itx_t, itx_lr) + itx->itx_lr.lrc_reclen);
list_destroy
(&clean_list);
itxg->itxg_txg != txg ?
* Locate the object node
and append its list.
End
Yes
No
t = &itxg->itxg_itxs->i_async_tree;
ian = avl_find(t, &oid, &where);
mutex_exit(&itxg
->itxg_lock);
ian != NULL ?
Yes
No
list_move_tail(&clean_
list, &ian->ia_list);
mutex_exit(&itxg
->itxg_lock);
txg++

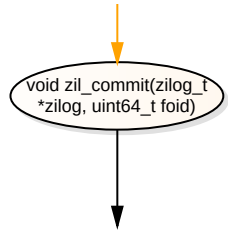

void zil_itx_assign (zilog_t *zilog, itx_t *itx, dmu_tx_t *tx)
uint64_t txg;
itxg_t *itxg;
itxs_t *itxs, *clean = NULL;
* Object ids can be re-instantiated in the next txg so
* remove any async transactions to avoid future leaks.
* This can happen if a fsync occurs on the re-instantiated
* object for a WR_INDIRECT or WR_NEED_COPY write, which gets
* the new file data and flushes a write record for the old object.
(itx->itx_lr.lrc_txtype & ~TX_CI) == TX_REMOVE ?
Yes
No
zil_remove_async(zilog, itx->itx_oid);
* Ensure the data of a renamed file is committed before the rename.
(itx->itx_lr.lrc_txtype & ~TX_CI) == TX_RENAME ?
Yes
No
zil_async_to_sync(zilog, itx->itx_oid);
spa_freeze_txg(zilog->zl_spa) != UINT64_MAX ?
Yes
No
txg = ZILTEST_TXG;
txg = dmu_tx_get_txg(tx);
itxg = &zilog->zl_itxg[txg & TXG_MASK];
mutex_enter(&itxg->itxg_lock);
itxs = itxg->itxg_itxs;
itxg->itxg_txg != txg ?
Yes
No
itxs != NULL ?
* The zil_clean callback hasn't got around to cleaning
* this itxg. Save the itxs for release below.
* This should be rare.
Yes
No
atomic_add_64(&zilog->zl_itx_list_sz, -itxg->itxg_sod);
itxg->itxg_sod = 0;
clean = itxg->itxg_itxs;
ASSERT(itxg->itxg_sod == 0);
itxg->itxg_txg = txg;
itxs = itxg->itxg_itxs = kmem_zalloc(sizeof (itxs_t), KM_SLEEP);
list_create(&itxs->i_sync_list, sizeof (itx_t), offsetof(itx_t, itx_node));
avl_create(&itxs->i_async_tree, zil_aitx_compare, sizeof (itx_async_node_t), offsetof(itx_async_node_t, ia_node));
itx->itx_sync ?
Yes
No
avl_tree_t *t = &itxs->i_async_tree;
uint64_t foid = ((lr_ooo_t *)&itx->itx_lr)->lr_foid;
itx_async_node_t *ian;
avl_index_t where;
ian = avl_find(t, &foid, &where);
list_insert_tail(&itxs->i_sync_list, itx);
atomic_add_64(&zilog->zl_itx_list_sz, itx->itx_sod);
itxg->itxg_sod += itx->itx_sod;
ian == NULL ?
No
Yes
ian = kmem_alloc(sizeof (itx_async_node_t), KM_SLEEP);
list_create(&ian->ia_list, sizeof (itx_t), offsetof(itx_t, itx_node));
ian->ia_foid = foid;
avl_insert(t, ian, where);
list_insert_tail(&ian->ia_list, itx);
itx->itx_lr.lrc_txg = dmu_tx_get_txg(tx);
zilog_dirty(zilog, txg);
mutex_exit(&itxg->itxg_lock);
Release the old itxs now we've dropped the lock
clean != NULL ?
Yes
No
zil_itxg_clean(clean);
End


void zil_clean(zilog_t *zilog, uint64_t synced_txg)
* If there are any in-memory intent log transactions which have now been
* synced then start up a taskq to free them. We should only do this after we
* have written out the uberblocks (i.e. txg has been comitted) so that
* don't inadvertently clean out in-memory log records that would be required
* by zil_commit().
itxg_t *itxg = &zilog->zl_itxg[synced_txg & TXG_MASK];
itxs_t *clean_me;
mutex_enter(&itxg->itxg_lock);
itxg->itxg_itxs == NULL || itxg->itxg_txg == ZILTEST_TXG ?
No
Yes
ASSERT3U(itxg->itxg_txg, <=, synced_txg);
ASSERT(itxg->itxg_txg != 0);
ASSERT(zilog->zl_clean_taskq != NULL);
atomic_add_64(&zilog->zl_itx_list_sz, -itxg->itxg_sod);
itxg->itxg_sod = 0;
clean_me = itxg->itxg_itxs;
itxg->itxg_itxs = NULL;
itxg->itxg_txg = 0;
mutex_exit(&itxg->itxg_lock);
* Preferably start a task queue to free up the old itxs but
* if taskq_dispatch can't allocate resources to do that then
* free it in-line. This should be rare. Note, using TQ_SLEEP
* created a bad performance problem.
mutex_exit(&itxg->itxg_lock);
taskq_dispatch(zilog->zl_clean_taskq, (void (*)(void *))zil_itxg_clean, clean_me, TQ_NOSLEEP) == NULL ?
Yes
No
zil_itxg_clean(clean_me);
End

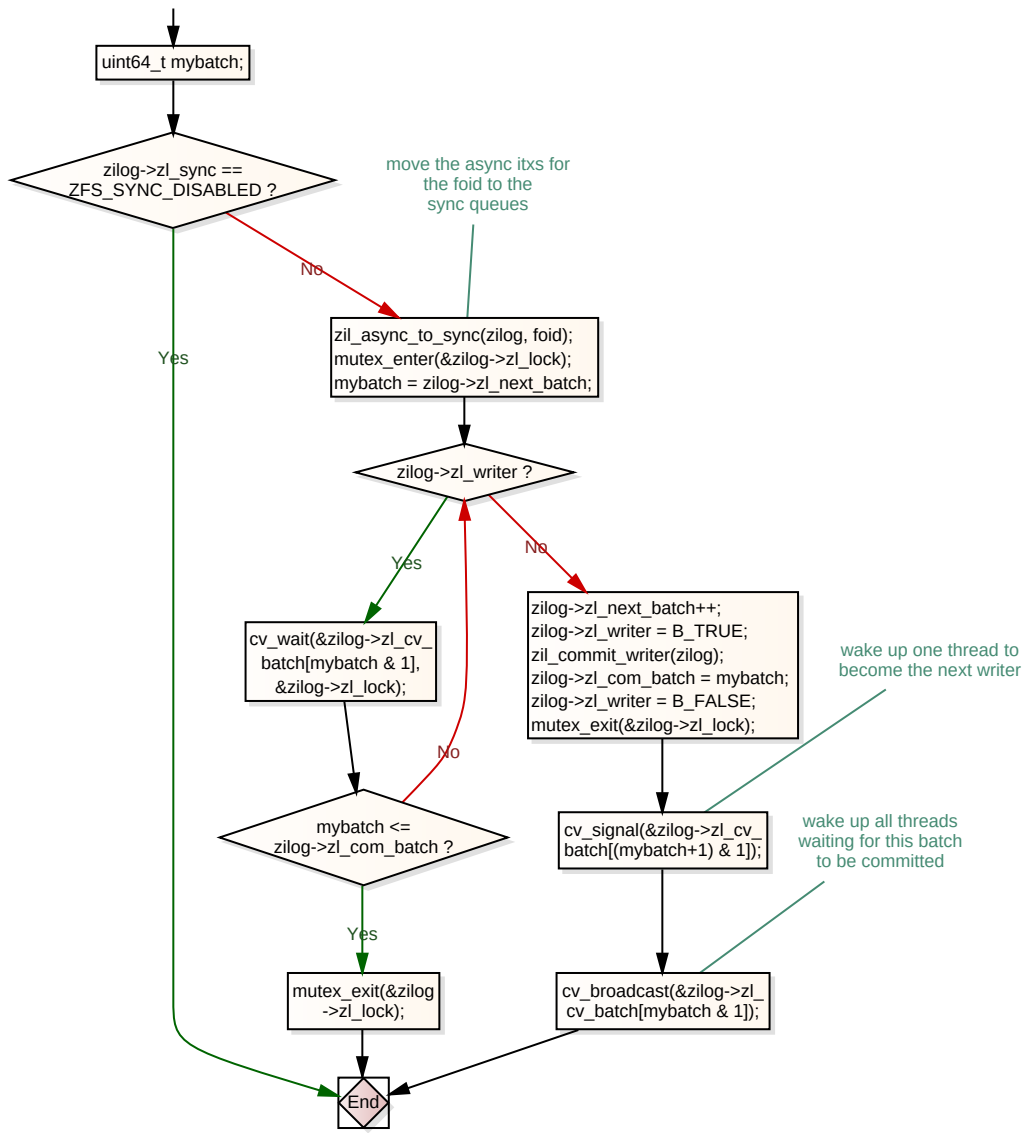

static void
zil_get_commit_list
(zilog_t *zilog)
* Get the list of itxs
to commit into
zl_itx_commit_list.
uint64_t otxg, txg;
list_t *commit_list = &zilog->zl_itx_commit_list;
uint64_t push_sod = 0;
ziltest support
spa_freeze_txg(zilog->zl_
spa) != UINT64_MAX ?
Yes
No
otxg = ZILTEST_TXG;
otxg = spa_last_synced_
txg(zilog->zl_spa) + 1;
txg = otxg
txg < (otxg +
TXG_CONCURRENT_STATES) ?
No
Yes
atomic_add_64(&zilog->zl_
itx_list_sz, -push_sod);
itxg_t *itxg = &zilog->zl_itxg[txg & TXG_MASK];
mutex_enter(&itxg->itxg_lock);
End
itxg->itxg_txg != txg ?
No
Yes
list_move_tail(commit_list, &itxg->itxg_itxs->i_sync_list);
push_sod += itxg->itxg_sod;
itxg->itxg_sod = 0;
mutex_exit(&itxg->itxg_lock);
mutex_exit(&itxg
->itxg_lock);
txg++

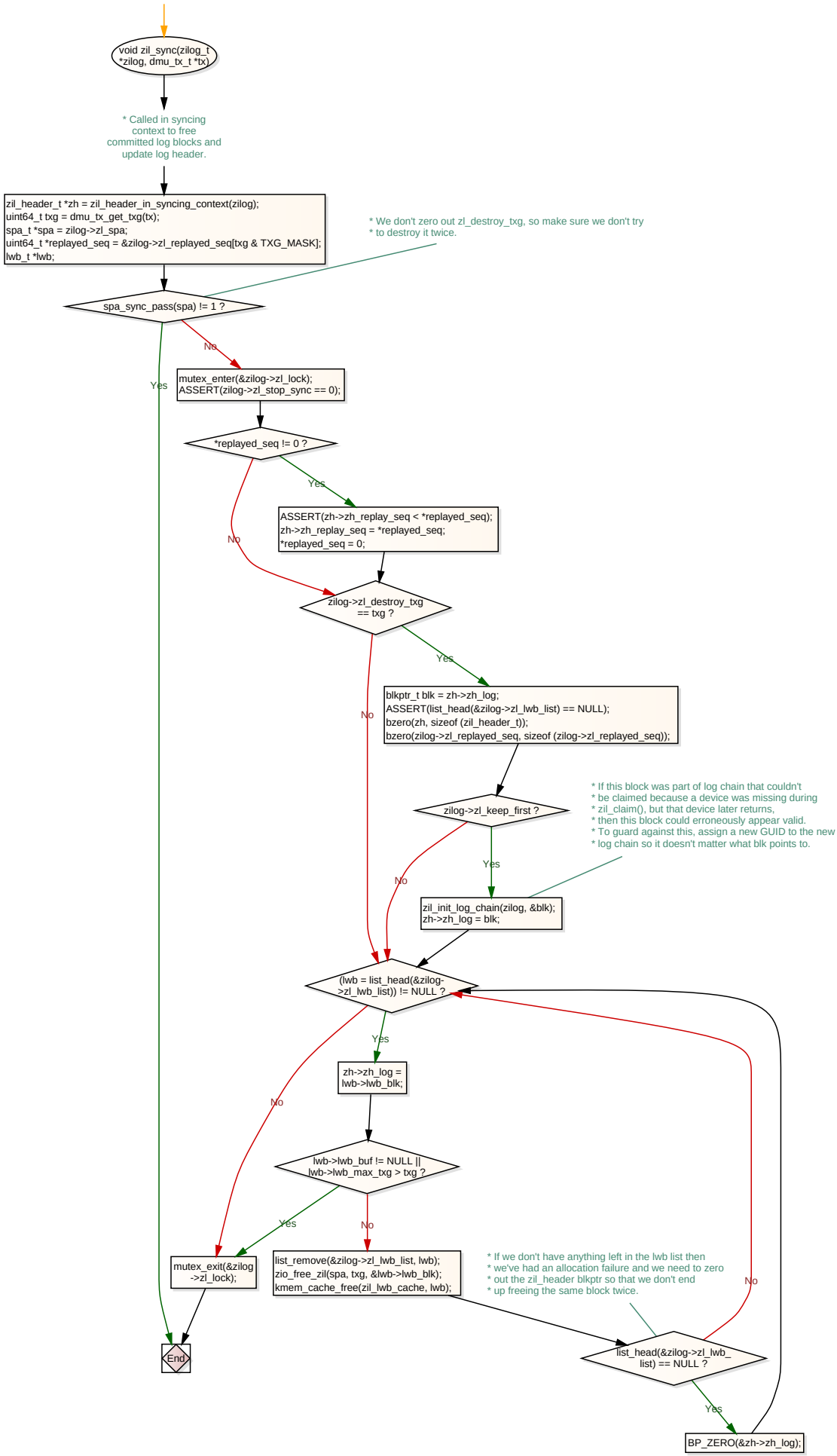

static void
zil_async_to_sync(zilog_t
*zilog, uint64_t foid)
* Move the async itxs
for a specified object
to commit into
sync lists.
uint64_t otxg, txg;
itx_async_node_t *ian;
avl_tree_t *t;
avl_index_t where;
ziltest support
spa_freeze_txg(zilog->zl_
spa) != UINT64_MAX ?
Yes
No
otxg = ZILTEST_TXG;
otxg = spa_last_synced_
txg(zilog->zl_spa) + 1;
txg = otxg
txg < (otxg +
TXG_CONCURRENT_STATES) ?
No
Yes
End
itxg_t *itxg = &zilog->zl_itxg[txg & TXG_MASK];
mutex_enter(&itxg->itxg_lock);
itxg->itxg_txg != txg ?
* If a foid is specified then find that node and append its
* list. Otherwise walk the tree appending all the lists
* to the sync list. We add to the end rather than the
* beginning to ensure the create has happened.
Yes
No
t = &itxg->itxg_itxs
->i_async_tree;
mutex_exit(&itxg
->itxg_lock);
foid != 0 ?
No
Yes
void *cookie = NULL;
ian = avl_find(t,
&foid, &where);
(ian = avl_destroy_nodes
(t, &cookie)) != NULL ?
ian != NULL ?
Yes
Yes
No
list_move_tail(&itxg->itxg_itxs->i_sync_list, &ian->ia_list);
list_destroy(&ian->ia_list);
kmem_free(ian, sizeof (itx_async_node_t));
list_move_tail(&itxg-
>itxg_itxs->i_sync_list,
&ian->ia_list);
No
mutex_exit(&itxg
->itxg_lock);
txg++

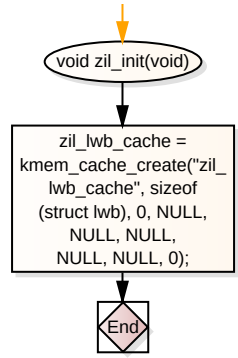

static void zil_commit_writer(zilog_t *zilog)
uint64_t txg;
itx_t *itx;
lwb_t *lwb;
spa_t *spa = zilog->zl_spa;
int error = 0;
ASSERT(zilog->zl_root_zio == NULL);
mutex_exit(&zilog->zl_lock);
zil_get_commit_list(zilog);
* Return if there's nothing to commit before we dirty the fs by
* calling zil_create().
list_head(&zilog->zl_itx_commit_list) == NULL ?
No
Yes
zilog->zl_suspend ?
Yes
No
mutex_enter(&zilog->zl_lock);
lwb = list_tail(&zilog->zl_lwb_list);
lwb = NULL;
lwb == NULL ?
No
Yes
lwb = zil_create(zilog);
DTRACE_PROBE1(zil__cw1, zilog_t *, zilog);
itx = list_head(&zilog->zl_itx_commit_list) ?
Yes
No
txg = itx->itx_lr.lrc_txg;
ASSERT(txg);
DTRACE_PROBE1(zil__cw2, zilog_t *, zilog);
write the last block out
txg > spa_last_synced_txg(spa) || txg > spa_freeze_txg(spa) ?
lwb != NULL && lwb->lwb_zio != NULL ?
Yes
No
Yes
No
lwb = zil_lwb_commit(zilog, itx, lwb);
lwb = zil_lwb_write_start(zilog, lwb);
list_remove(&zilog->zl_itx_commit_list, itx);
kmem_free(itx, offsetof(itx_t, itx_lr) + itx->itx_lr.lrc_reclen);
zilog->zl_cur_used = 0;
* Wait if necessary for the log blocks to be on stable storage.
zilog->zl_root_zio ?
No
Yes
error = zio_wait(zilog->zl_root_zio);
zilog->zl_root_zio = NULL;
zil_flush_vdevs(zilog);
error || lwb == NULL ?
No
Yes
txg_wait_synced(zilog->zl_dmu_pool, 0);
mutex_enter(&zilog->zl_lock);
* Remember the highest committed log sequence number for ztest.
* We only update this value when all the log writes succeeded,
* because ztest wants to ASSERT that it got the whole log chain.
error == 0 && lwb != NULL ?
Yes
No
zilog->zl_commit_lr_seq = zilog->zl_lr_seq;
End

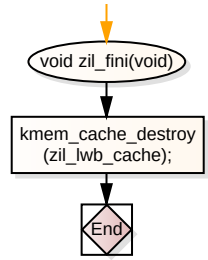


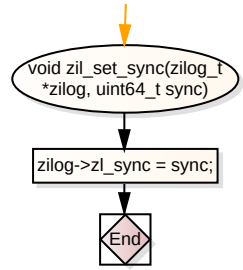

```
* Commit zfs transactions to stable storage.
* If foid is 0 push out all transactions, otherwise push only those
* for that object or might reference that object.
*
* itxs are committed in batches. In a heavily stressed zil there will be
* a commit writer thread who is writing out a bunch of itxs to the log
* for a set of committing threads (cthreads) in the same batch as the writer.
* Those cthreads are all waiting on the same cv for that batch.
*
* There will also be a different and growing batch of threads that are
* waiting to commit (qthreads). When the committing batch completes
* a transition occurs such that the cthreads exit and the qthreads become
* cthreads. One of the new cthreads becomes the writer thread for the
* batch. Any new threads arriving become new qthreads.
*
* Only 2 condition variables are needed and there's no transition
* between the two cvs needed. They just flip-flop between qthreads
* and cthreads.
*
* Using this scheme we can efficiently wakeup up only those threads
* that have been committed.
```

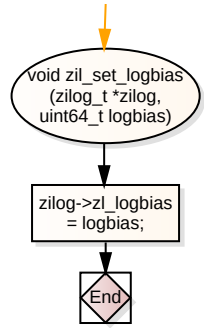


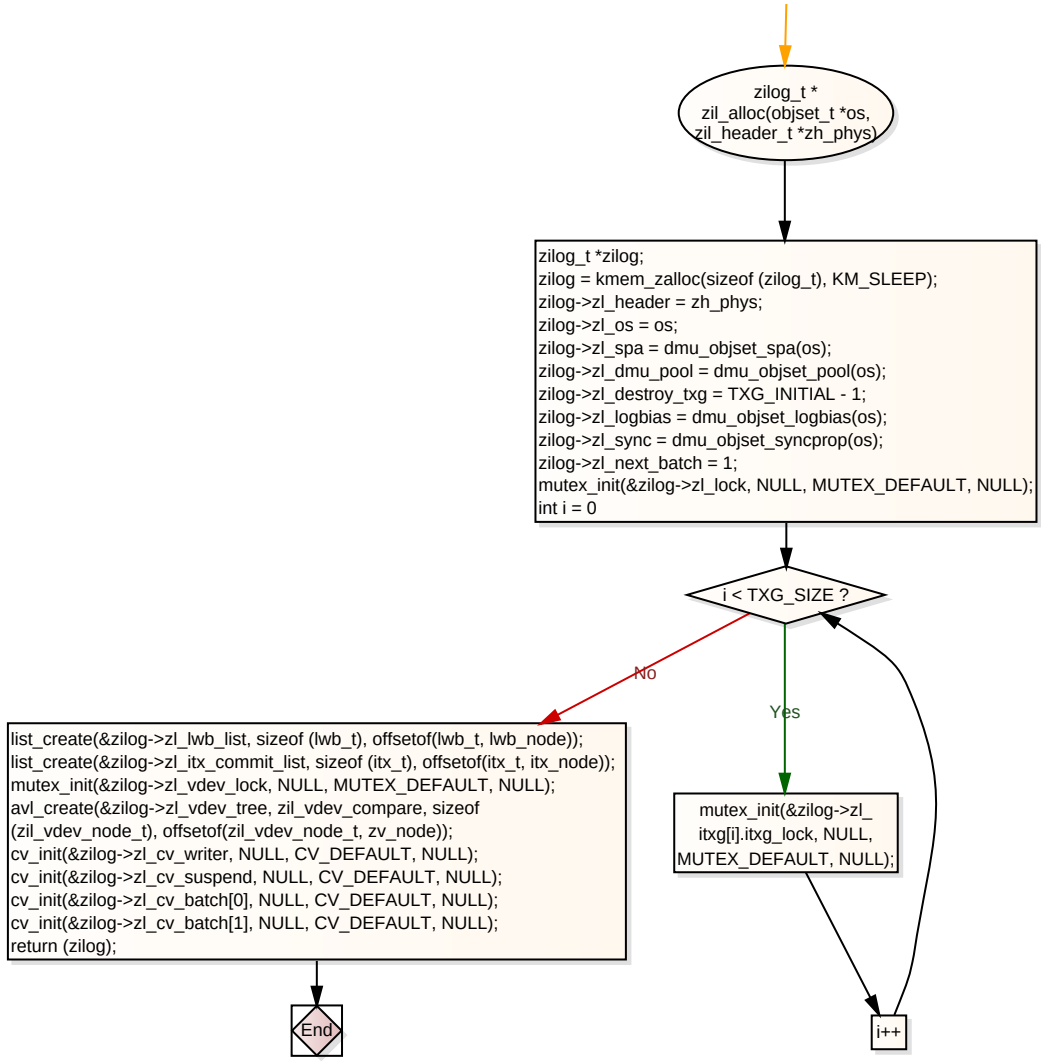

void zil_sync(zilog_t *zilog, dmu_tx_t *tx)
* Called in syncing context to free committed log blocks and update log header.
zil_header_t *zh = zil_header_in_syncing_context(zilog);
uint64_t txg = dmu_tx_get_txg(tx);
spa_t *spa = zilog->zl_spa;
uint64_t *replayed_seq = &zilog->zl_replayed_seq[txg & TXG_MASK];
lwb_t *lwb;
* We don't zero out zl_destroy_txg, so make sure we don't try * to destroy it twice.
spa_sync_pass(spa) != 1 ?
No
Yes
mutex_enter(&zilog->zl_lock);
ASSERT(zilog->zl_stop_sync == 0);
*replayed_seq != 0 ?
Yes
No
ASSERT(zh->zh_replay_seq < *replayed_seq);
zh->zh_replay_seq = *replayed_seq;
*replayed_seq = 0;
zilog->zl_destroy_txg == txg ?
Yes
No
blkptr_t blk = zh->zh_log;
ASSERT(list_head(&zilog->zl_lwb_list) == NULL);
bzero(zh, sizeof (zil_header_t));
bzero(zilog->zl_replayed_seq, sizeof (zilog->zl_replayed_seq));
zilog->zl_keep_first ?
* If this block was part of log chain that couldn't * be claimed because a device was missing during * zil_claim(), but that device later returns, * then this block could erroneously appear valid. * To guard against this, assign a new GUID to the new * log chain so it doesn't matter what blk points to.
Yes
No
zil_init_log_chain(zilog, &blk);
zh->zh_log = blk;
(lwb = list_head(&zilog->zl_lwb_list)) != NULL ?
Yes
No
zh->zh_log = lwb->lwb_blk;
lwb->lwb_buf != NULL || lwb->lwb_max_txg > txg ?
Yes
No
mutex_exit(&zilog->zl_lock);
list_remove(&zilog->zl_lwb_list, lwb);
zio_free_zil(spa, txg, &lwb->lwb_blk);
kmem_cache_free(zil_lwb_cache, lwb);
* If we don't have anything left in the lwb list then * we've had an allocation failure and we need to zero * out the zil_header blkptr so that we don't end * up freeing the same block twice.
list_head(&zilog->zl_lwb_list) == NULL ?
No
Yes
BP_ZERO(&zh->zh_log);
End

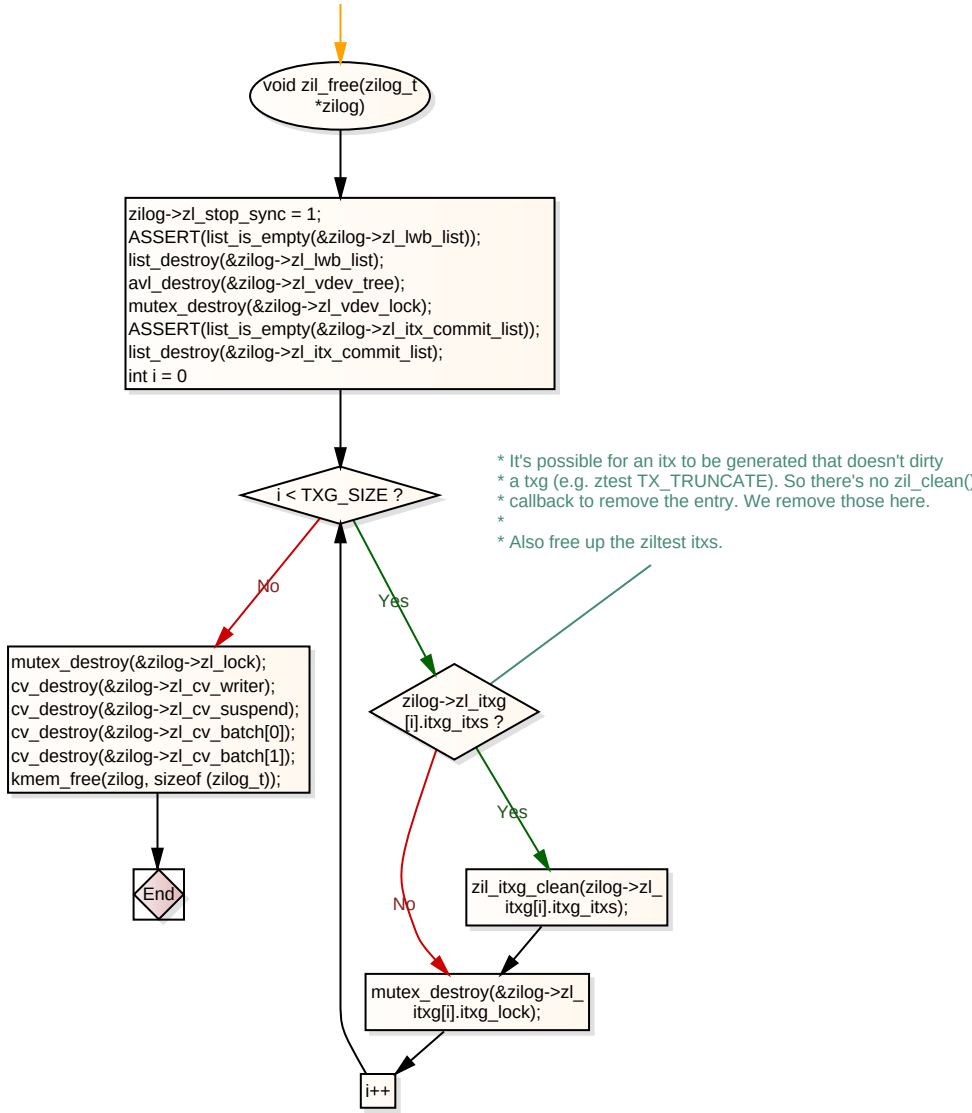

void zil_init(void)
zil_lwb_cache = kmem_cache_create("zil_ lwb_cache", sizeof (struct lwb), 0, NULL, NULL, NULL, NULL, NULL, 0);
End

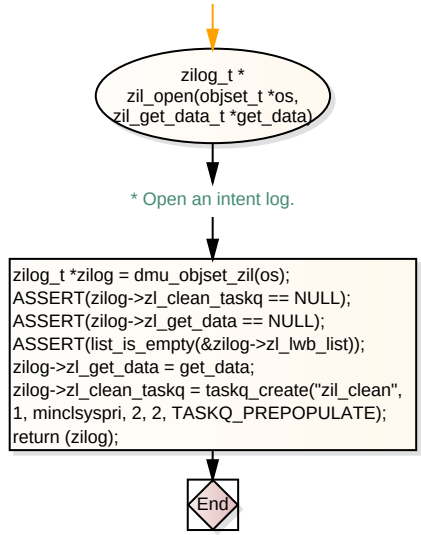

void zil_fini(void)
kmem_cache_destroy
(zil_lwb_cache);
End

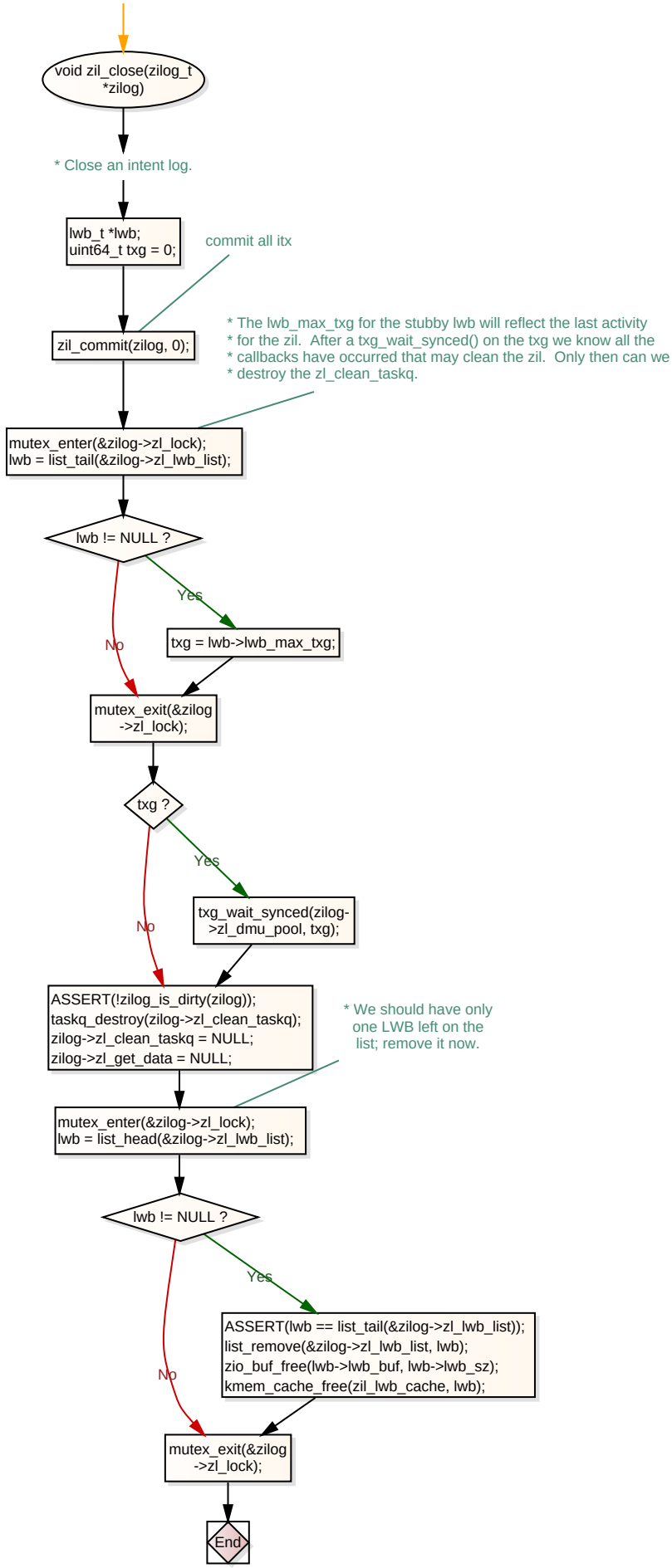

void zil_set_sync(zilog_t *zilog, uint64_t sync)
zilog->zl_sync = sync;
End

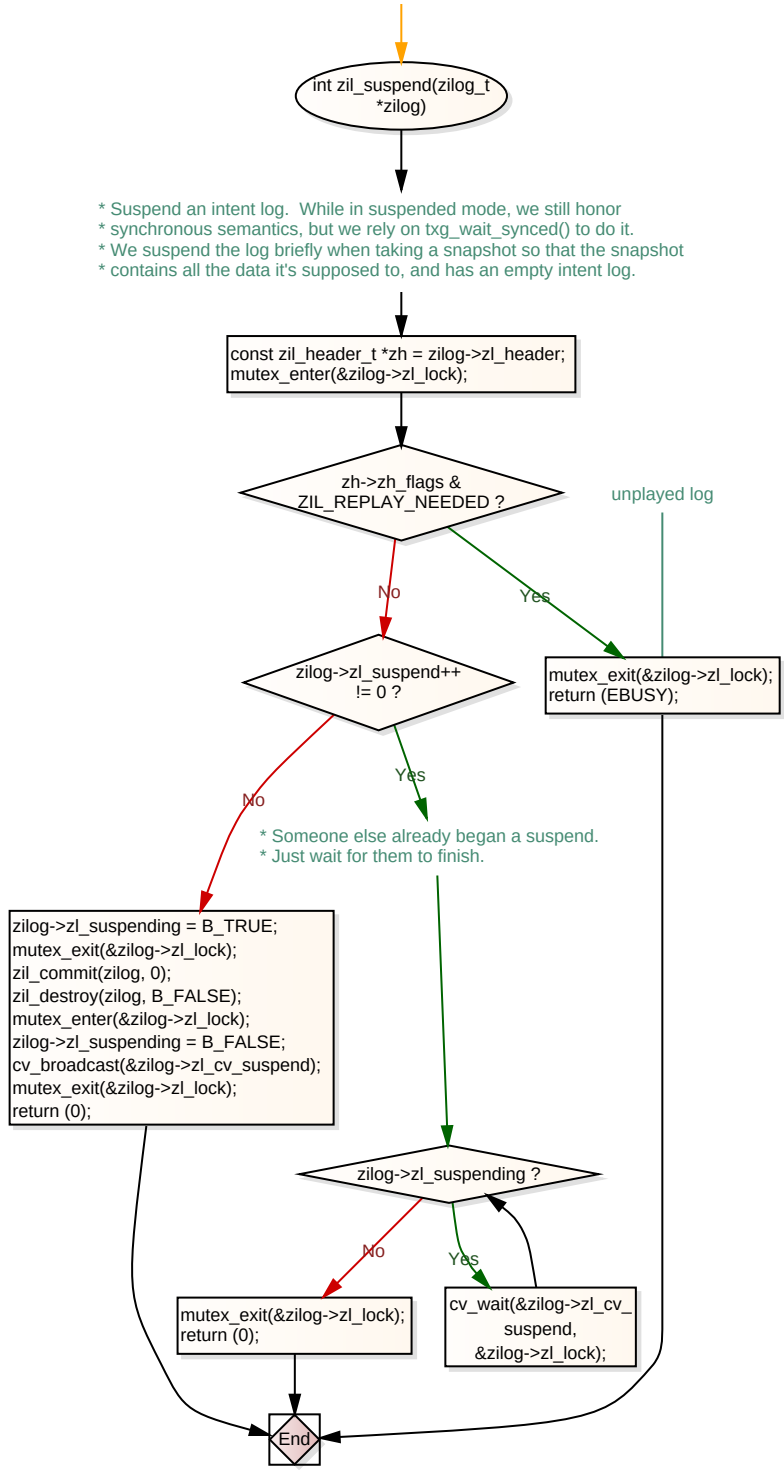

void zil_set_logbias
(zilog_t *zilog,
uint64_t logbias)
zilog->zl_logbias
= logbias;
End

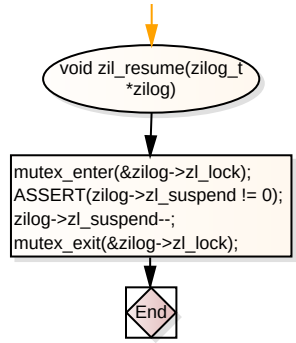

zilog_t *
zil_alloc(objset_t *os,
zil_header_t *zh_phys)
zilog_t *zilog;
zilog = kmem_zalloc(sizeof (zilog_t), KM_SLEEP);
zilog->zl_header = zh_phys;
zilog->zl_os = os;
zilog->zl_spa = dmu_objset_spa(os);
zilog->zl_dmu_pool = dmu_objset_pool(os);
zilog->zl_destroy_txg = TXG_INITIAL - 1;
zilog->zl_logbias = dmu_objset_logbias(os);
zilog->zl_sync = dmu_objset_syncprop(os);
zilog->zl_next_batch = 1;
mutex_init(&zilog->zl_lock, NULL, MUTEX_DEFAULT, NULL);
int i = 0
i < TXG_SIZE ?
No
Yes
list_create(&zilog->zl_lwb_list, sizeof (lwb_t), offsetof(lwb_t, lwb_node));
list_create(&zilog->zl_itx_commit_list, sizeof (itx_t), offsetof(itx_t, itx_node));
mutex_init(&zilog->zl_vdev_lock, NULL, MUTEX_DEFAULT, NULL);
avl_create(&zilog->zl_vdev_tree, zil_vdev_compare, sizeof
(zil_vdev_node_t), offsetof(zil_vdev_node_t, zv_node));
cv_init(&zilog->zl_cv_writer, NULL, CV_DEFAULT, NULL);
cv_init(&zilog->zl_cv_suspend, NULL, CV_DEFAULT, NULL);
cv_init(&zilog->zl_cv_batch[0], NULL, CV_DEFAULT, NULL);
cv_init(&zilog->zl_cv_batch[1], NULL, CV_DEFAULT, NULL);
return (zilog);
mutex_init(&zilog->zl_
itxg[i].itxg_lock, NULL,
MUTEX_DEFAULT, NULL);
End
i++

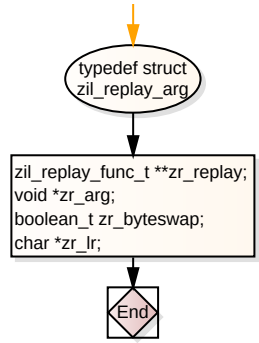

void zil_free(zilog_t *zilog)
zilog->zl_stop_sync = 1;
ASSERT(list_is_empty(&zilog->zl_lwb_list));
list_destroy(&zilog->zl_lwb_list);
avl_destroy(&zilog->zl_vdev_tree);
mutex_destroy(&zilog->zl_vdev_lock);
ASSERT(list_is_empty(&zilog->zl_itx_commit_list));
list_destroy(&zilog->zl_itx_commit_list);
int i = 0
i < TXG_SIZE ?
No
Yes
* It's possible for an itx to be generated that doesn't dirty
* a txg (e.g. ztest TX_TRUNCATE). So there's no zil_clean()
* callback to remove the entry. We remove those here.
*
* Also free up the ziltest itxs.
mutex_destroy(&zilog->zl_lock);
cv_destroy(&zilog->zl_cv_writer);
cv_destroy(&zilog->zl_cv_suspend);
cv_destroy(&zilog->zl_cv_batch[0]);
cv_destroy(&zilog->zl_cv_batch[1]);
kmem_free(zilog, sizeof (zilog_t));
zilog->zl_itxg [i].itxg_itxs ?
Yes
No
End
zil_itxg_clean(zilog->zl_ itxg[i].itxg_itxs);
mutex_destroy(&zilog->zl_ itxg[i].itxg_lock);
i++

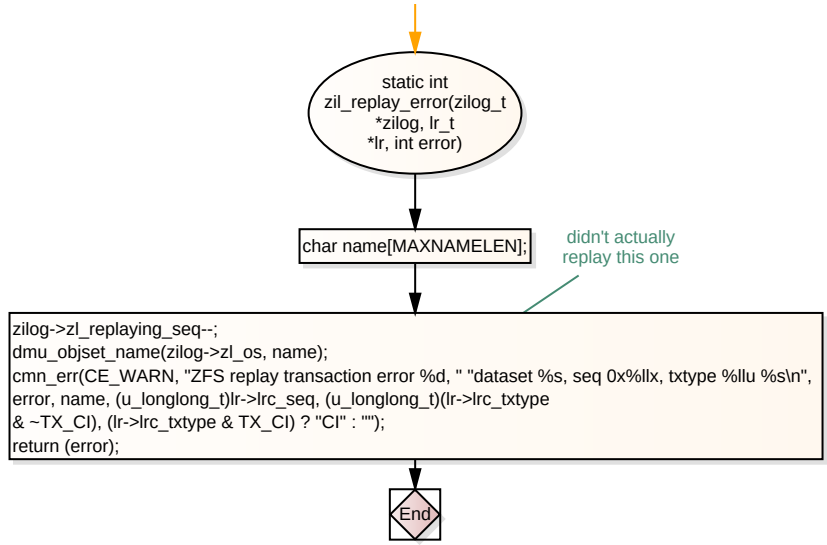

zilog_t *
zil_open(objset_t *os,
zil_get_data_t *get_data)
* Open an intent log.
zilog_t *zilog = dmu_objset_zil(os);
ASSERT(zilog->zl_clean_taskq == NULL);
ASSERT(zilog->zl_get_data == NULL);
ASSERT(list_is_empty(&zilog->zl_lwb_list));
zilog->zl_get_data = get_data;
zilog->zl_clean_taskq = taskq_create("zil_clean",
1, minclsyspri, 2, 2, TASKQ_PREPOPULATE);
return (zilog);
End

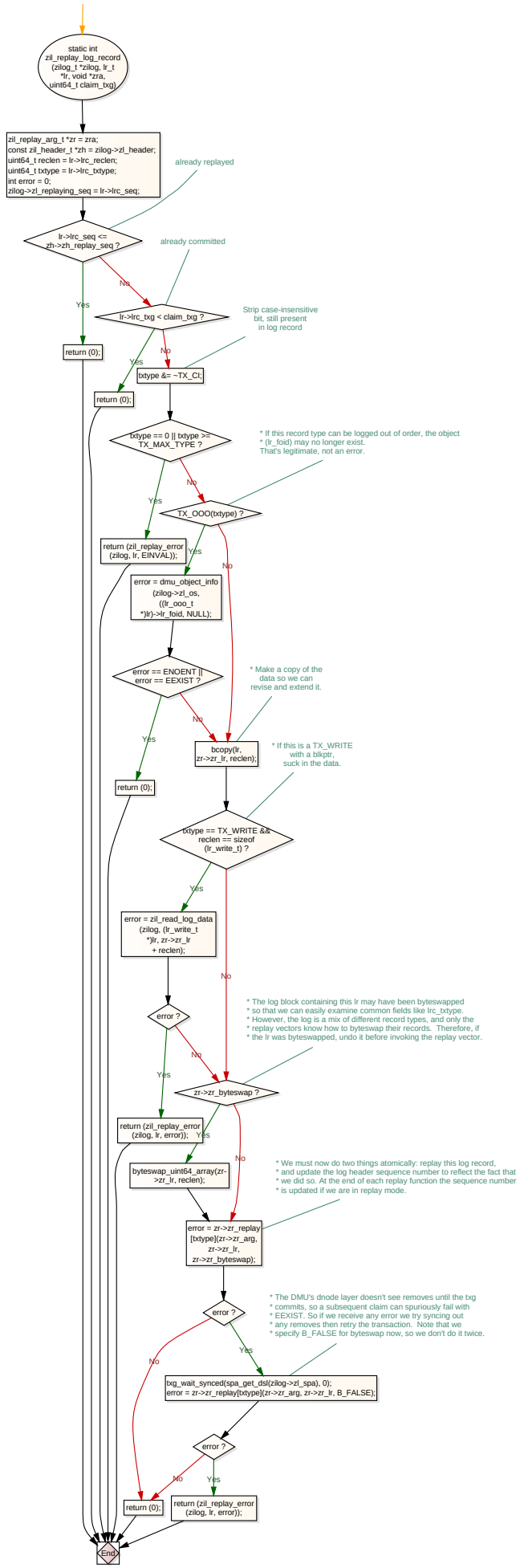

void zil_close(zilog_t *zilog)
* Close an intent log.
lwb_t *lwb;
uint64_t txg = 0;
commit all itx
zil_commit(zilog, 0);
* The lwb_max_txg for the stubby lwb will reflect the last activity
* for the zil. After a txg_wait_synced() on the txg we know all the
* callbacks have occurred that may clean the zil. Only then can we
* destroy the zl_clean_taskq.
mutex_enter(&zilog->zl_lock);
lwb = list_tail(&zilog->zl_lwb_list);
lwb != NULL ?
Yes
No
txg = lwb->lwb_max_txg;
mutex_exit(&zilog ->zl_lock);
txg ?
Yes
No
txg_wait_synced(zilog- >zl_dmu_pool, txg);
ASSERT(!zilog_is_dirty(zilog));
taskq_destroy(zilog->zl_clean_taskq);
zilog->zl_clean_taskq = NULL;
zilog->zl_get_data = NULL;
* We should have only one LWB left on the list; remove it now.
mutex_enter(&zilog->zl_lock);
lwb = list_head(&zilog->zl_lwb_list);
lwb != NULL ?
Yes
No
ASSERT(lwb == list_tail(&zilog->zl_lwb_list));
list_remove(&zilog->zl_lwb_list, lwb);
zio_buf_free(lwb->lwb_buf, lwb->lwb_sz);
kmem_cache_free(zil_lwb_cache, lwb);
mutex_exit(&zilog ->zl_lock);
End

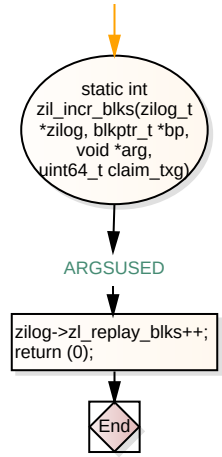

int zil_suspend(zilog_t *zilog)
* Suspend an intent log. While in suspended mode, we still honor
* synchronous semantics, but we rely on txg_wait_synced() to do it.
* We suspend the log briefly when taking a snapshot so that the snapshot
* contains all the data it's supposed to, and has an empty intent log.
const zil_header_t *zh = zilog->zl_header;
mutex_enter(&zilog->zl_lock);
zh->zh_flags & ZIL_REPLAY_NEEDED ?
unplayed log
No
Yes
zilog->zl_suspend++ != 0 ?
mutex_exit(&zilog->zl_lock);
return (EBUSY);
No
Yes
* Someone else already began a suspend.
* Just wait for them to finish.
zilog->zl_suspending = B_TRUE;
mutex_exit(&zilog->zl_lock);
zil_commit(zilog, 0);
zil_destroy(zilog, B_FALSE);
mutex_enter(&zilog->zl_lock);
zilog->zl_suspending = B_FALSE;
cv_broadcast(&zilog->zl_cv_suspend);
mutex_exit(&zilog->zl_lock);
return (0);
zilog->zl_suspending ?
No
Yes
mutex_exit(&zilog->zl_lock);
return (0);
cv_wait(&zilog->zl_cv_suspend, &zilog->zl_lock);
End

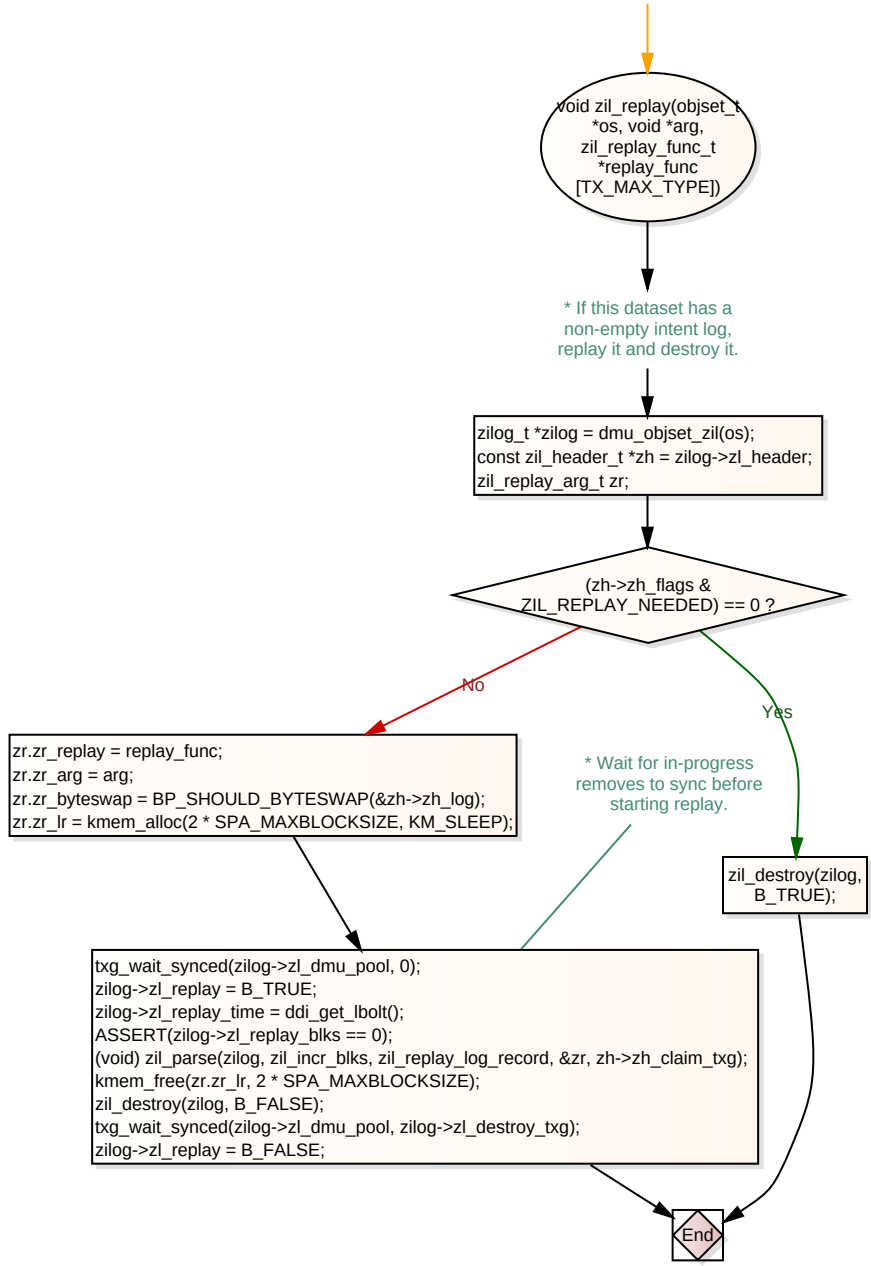

void zil_resume(zilog_t *zilog)
mutex_enter(&zilog->zl_lock);
ASSERT(zilog->zl_suspend != 0);
zilog->zl_suspend--;
mutex_exit(&zilog->zl_lock);
End

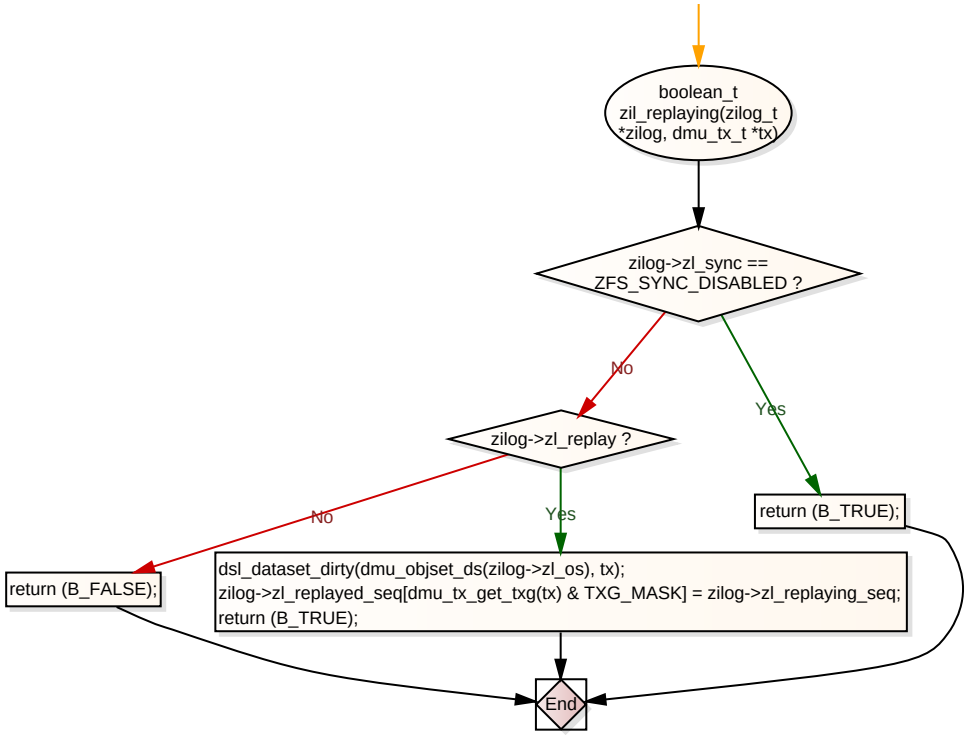

typedef struct
zil_replay_arg
zil_replay_func_t **zr_replay;
void *zr_arg;
boolean_t zr_byteswap;
char *zr_lr;
End

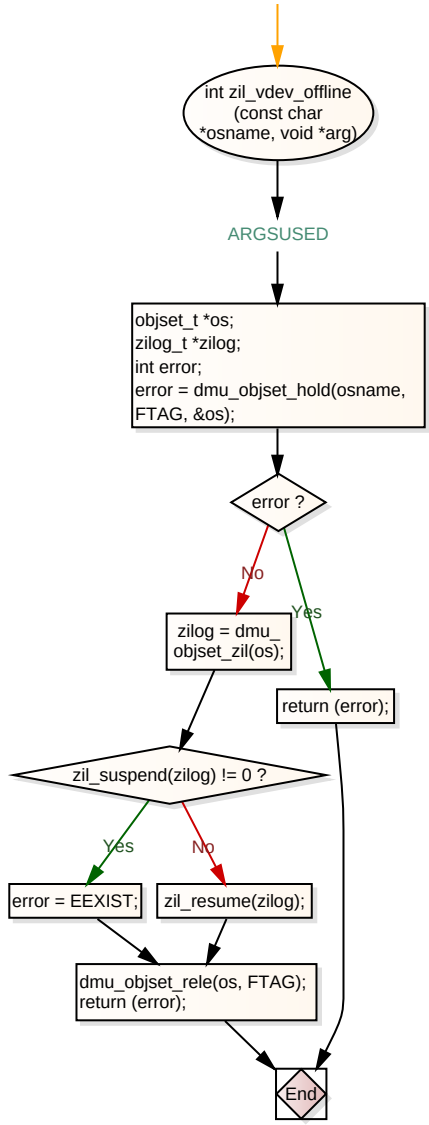

static int
zil_replay_error(zilog_t
*zilog, lr_t
*lr, int error)
char name[MAXNAMELEN];
didn't actually
replay this one
zilog->zl_replaying_seq--;
dmu_objset_name(zilog->zl_os, name);
cmn_err(CE_WARN, "ZFS replay transaction error %d, " "dataset %s, seq 0x%llx, txtype %llu %s\n",
error, name, (u_longlong_t)lr->lrc_seq, (u_longlong_t)(lr->lrc_txtype
& ~TX_CI), (lr->lrc_txtype & TX_CI) ? "CI" : "");
return (error);
End


static int
zil_replay_log_record
(zilog_t *zilog, lr_t
*lr, void *zra,
uint64_t claim_txg)
zil_replay_arg_t *zr = zra;
const zil_header_t *zh = zilog->zl_header;
uint64_t reclen = lr->lrc_reclen;
uint64_t txtype = lr->lrc_txtype;
int error = 0;
zilog->zl_replaying_seq = lr->lrc_seq;
already replayed
lr->lrc_seq <=
zh->zh_replay_seq ?
Yes
No
already committed
lr->lrc_txg < claim_txg ?
Yes
No
Strip case-insensitive
bit, still present
in log record
return (0);
txtype &= ~TX_CI;
return (0);
txtype == 0 || txtype >=
TX_MAX_TYPE ?
Yes
No
* If this record type can be logged out of order, the object
* (lr_foid) may no longer exist.
That's legitimate, not an error.
TX_OOO(txtype) ?
Yes
No
return (zil_replay_error
(zilog, lr, EINVAL));
error = dmu_object_info
(zilog->zl_os,
((lr_ooo_t
*)lr)->lr_foid, NULL);
error == ENOENT ||
error == EEXIST ?
Yes
No
* Make a copy of the
data so we can
revise and extend it.
return (0);
bcopy(lr,
zr->zr_lr, reclen);
* If this is a TX_WRITE
with a blkptr,
suck in the data.
txtype == TX_WRITE &&
reclen == sizeof
(lr_write_t) ?
Yes
No
error = zil_read_log_data
(zilog, (lr_write_t
*)lr, zr->zr_lr
+ reclen);
error ?
Yes
No
* The log block containing this lr may have been byteswapped
* so that we can easily examine common fields like lrc_txtype.
* However, the log is a mix of different record types, and only the
* replay vectors know how to byteswap their records. Therefore, if
* the lr was byteswapped, undo it before invoking the replay vector.
zr->zr_byteswap ?
Yes
No
return (zil_replay_error
(zilog, lr, error));
byteswap_uint64_array(zr-
>zr_lr, reclen);
* We must now do two things atomically: replay this log record,
* and update the log header sequence number to reflect the fact that
* we did so. At the end of each replay function the sequence number
* is updated if we are in replay mode.
error = zr->zr_replay
[txtype](zr->zr_arg,
zr->zr_lr,
zr->zr_byteswap);
error ?
Yes
No
* The DMU's dnode layer doesn't see removes until the txg
* commits, so a subsequent claim can spuriously fail with
* EEXIST. So if we receive any error we try syncing out
* any removes then retry the transaction. Note that we
* specify B_FALSE for byteswap now, so we don't do it twice.
txg_wait_synced(spa_get_dsl(zilog->zl_spa), 0);
error = zr->zr_replay[txtype](zr->zr_arg, zr->zr_lr, B_FALSE);
error ?
No
Yes
return (0);
return (zil_replay_error
(zilog, lr, error));
End


static int
zil_incr_blks(zilog_t
*zilog, blkptr_t *bp,
void *arg,
uint64_t claim_txg)
ARGSUSED
zilog->zl_replay_blks++;
return (0);
End


void zil_replay(objset_t *os, void *arg, zil_replay_func_t *replay_func [TX_MAX_TYPE])
* If this dataset has a non-empty intent log, replay it and destroy it.
zilog_t *zilog = dmu_objset_zil(os);
const zil_header_t *zh = zilog->zl_header;
zil_replay_arg_t zr;
(zh->zh_flags & ZIL_REPLAY_NEEDED) == 0 ?
No
Yes
zr.zr_replay = replay_func;
zr.zr_arg = arg;
zr.zr_byteswap = BP_SHOULD_BYTESWAP(&zh->zh_log);
zr.zr_lr = kmem_alloc(2 * SPA_MAXBLOCKSIZE, KM_SLEEP);
* Wait for in-progress removes to sync before starting replay.
zil_destroy(zilog, B_TRUE);
txg_wait_synced(zilog->zl_dmu_pool, 0);
zilog->zl_replay = B_TRUE;
zilog->zl_replay_time = ddi_get_lbolt();
ASSERT(zilog->zl_replay_blks == 0);
(void) zil_parse(zilog, zil_incr_blks, zil_replay_log_record, &zr, zh->zh_claim_txg);
kmem_free(zr.zr_lr, 2 * SPA_MAXBLOCKSIZE);
zil_destroy(zilog, B_FALSE);
txg_wait_synced(zilog->zl_dmu_pool, zilog->zl_destroy_txg);
zilog->zl_replay = B_FALSE;
End


boolean_t
zil_replaying(zilog_t
*zilog, dmu_tx_t *tx)
zilog->zl_sync ==
ZFS_SYNC_DISABLED ?
No
Yes
zilog->zl_replay ?
No
Yes
return (B_TRUE);
return (B_FALSE);
dsl_dataset_dirty(dmu_objset_ds(zilog->zl_os), tx);
zilog->zl_replayed_seq[dmu_tx_get_txg(tx) & TXG_MASK] = zilog->zl_replaying_seq;
return (B_TRUE);
End


int zil_vdev_offline
(const char
*osname, void *arg)
ARGSUSED
objset_t *os;
zilog_t *zilog;
int error;
error = dmu_objset_hold(osname,
FTAG, &os);
error ?
No
Yes
zilog = dmu_
objset_zil(os);
return (error);
zil_suspend(zilog) != 0 ?
Yes
No
error = EEXIST;
zil_resume(zilog);
dmu_objset_rele(os, FTAG);
return (error);
End