```
**********************************************************
    7548 Wed Jun 11 19:45:18 2014
new/usr/src/cmd/sgs/rtld/common/malloc.c
XXXX all calloc() implementations should check overflow
**********************************************************
______unchanged_portion_omitted_

void *
calloc(size_t num, size_t size)
{
        void *  mp;
        size_t  total;
#endif /* ! codereview */

        if (num == 0 || size == 0) {
                total = 0;
        } else {
                total = num * size;

                /* check for overflow */
                if (total / num != size) {
                        errno = ENOMEM;
        num *= size;
        if ((mp = malloc(num)) == NULL)
                        return (NULL);
                }
        }

        if ((mp = malloc(total)) == NULL)
                return (NULL);
        (void) memset(mp, 0, total);
        (void) memset(mp, 0, num);
        return (mp);
}
______unchanged_portion_omitted_
```

```
**********************************************************
   31831 Wed Jun 11 19:45:19 2014
new/usr/src/lib/libmalloc/common/malloc.c
XXXX all calloc() implementations should check overflow
**********************************************************
/*
 * 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 2008 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 */

/*	Copyright (c) 1988 AT&T	*/
/*	  All Rights Reserved  	*/

#pragma ident	"%Z%%M%	%I%	%E% SMI"

#include <sys/types.h>

#ifndef debug
#define	NDEBUG
#endif

#include <stdlib.h>
#include <string.h>
#include <errno.h>
#endif /* ! codereview */
#include "assert.h"
#include "malloc.h"
#include "mallint.h"
#include <thread.h>
#include <pthread.h>
#include <synch.h>
#include <unistd.h>
#include <limits.h>

static mutex_t mlock = DEFAULTMUTEX;
static ssize_t freespace(struct holdblk *);
static void *malloc_unlocked(size_t, int);
static void *realloc_unlocked(void *, size_t);
static void free_unlocked(void *);
static void *morecore(size_t);

/*
 * use level memory allocater (malloc, free, realloc)
 *
 *	-malloc, free, realloc and mallopt form a memory allocator
```

```
 *	similar to malloc, free, and realloc.  The routines
 *	here are much faster than the original, with slightly worse
 *	space usage (a few percent difference on most input).  They
 *	do not have the property that data in freed blocks is left
 *	untouched until the space is reallocated.
 *
 *	-Memory is kept in the "arena", a singly linked list of blocks.
 *	These blocks are of 3 types.
 *		1. A free block.  This is a block not in use by the
 *		   user.  It has a 3 word header. (See description
 *		   of the free queue.)
 *		2. An allocated block.  This is a block the user has
 *		   requested.  It has only a 1 word header, pointing
 *		   to the next block of any sort.
 *		3. A permanently allocated block.  This covers space
 *		   aquired by the user directly through sbrk().  It
 *		   has a 1 word header, as does 2.
 *	Blocks of type 1 have the lower bit of the pointer to the
 *	nextblock = 0.  Blocks of type 2 and 3 have that bit set,
 *	to mark them busy.
 *
 *	-Unallocated blocks are kept on an unsorted doubly linked
 *	free list.
 *
 *	-Memory is allocated in blocks, with sizes specified by the
 *	user.  A circular first-fit startegy is used, with a roving
 *	head of the free queue, which prevents bunching of small
 *	blocks at the head of the queue.
 *
 *	-Compaction is performed at free time of any blocks immediately
 *	following the freed block.  The freed block will be combined
 *	with a preceding block during the search phase of malloc.
 *	Since a freed block is added at the front of the free queue,
 *	which is moved to the end of the queue if considered and
 *	rejected during the search, fragmentation only occurs if
 *	a block with a contiguious preceding block that is free is
 *	freed and reallocated on the next call to malloc.  The
 *	time savings of this strategy is judged to be worth the
 *	occasional waste of memory.
 *
 *	-Small blocks (of size < MAXSIZE)  are not allocated directly.
 *	A large "holding" block is allocated via a recursive call to
 *	malloc.  This block contains a header and ?????? small blocks.
 *	Holding blocks for a given size of small block (rounded to the
 *	nearest ALIGNSZ bytes) are kept on a queue with the property that any
 *	holding block with an unused small block is in front of any without.
 *	A list of free blocks is kept within the holding block.
 */

/*
 *	description of arena, free queue, holding blocks etc.
 *
 * New compiler and linker does not guarentee order of initialized data.
 * Define freeptr as arena[2-3] to guarentee it follows arena in memory.
 * Later code depends on this order.
 */

static struct header arena[4] = {
	    {0, 0, 0},
	    {0, 0, 0},
	    {0, 0, 0},
	    {0, 0, 0}
	};
				/*
				 * the second word is a minimal block to
				 * start the arena. The first is a busy
```

```
                                 * block to be pointed to by the last block.
                                 */

#define freeptr (arena + 2)
                                /* first and last entry in free list */
static struct header *arenaend; /* ptr to block marking high end of arena */
static struct header *lastblk;  /* the highest block in the arena */
static struct holdblk **holdhead;   /* pointer to array of head pointers */
                                    /* to holding block chains */
/*
 * In order to save time calculating indices, the array is 1 too
 * large, and the first element is unused
 *
 * Variables controlling algorithm, esp. how holding blocs are used
 */
static int numlblks = NUMLBLKS;
static int minhead = MINHEAD;
static int change = 0;  /* != 0, once param changes are no longer allowed */
static int fastct = FASTCT;
static unsigned int maxfast = MAXFAST;
/* number of small block sizes to map to one size */

static int grain = ALIGNSZ;

#ifdef debug
static int case1count = 0;

static void
checkq(void)
{
        register struct header *p;

        p = &freeptr[0];

        /* check forward */
        /*CSTYLED*/
        while (p != &freeptr[1]) {
                p = p->nextfree;
                assert(p->prevfree->nextfree == p);
        }

        /* check backward */
        /*CSTYLED*/
        while (p != &freeptr[0]) {
                p = p->prevfree;
                assert(p->nextfree->prevfree == p);
        }
}
#endif


/*
 * malloc(nbytes) - give a user nbytes to use
 */

void *
malloc(size_t nbytes)
{
        void *ret;

        (void) mutex_lock(&mlock);
        ret = malloc_unlocked(nbytes, 0);
        (void) mutex_unlock(&mlock);
        return (ret);
}
```

```
/*
 * Use malloc_unlocked() to get the address to start with; Given this
 * address, find out the closest address that aligns with the request
 * and return that address after doing some house keeping (refer to the
 * ascii art below).
 */
void *
memalign(size_t alignment, size_t size)
{
        void *alloc_buf;
        struct header *hd;
        size_t alloc_size;
        uintptr_t fr;
        static int realloc;

        if (size == 0 || alignment == 0 ||
            (alignment & (alignment - 1)) != 0) {
                return (NULL);
        }
        if (alignment <= ALIGNSZ)
                return (malloc(size));

        alloc_size = size + alignment;
        if (alloc_size < size) { /* overflow */
                return (NULL);
        }

        (void) mutex_lock(&mlock);
        alloc_buf = malloc_unlocked(alloc_size, 1);
        (void) mutex_unlock(&mlock);

        if (alloc_buf == NULL)
                return (NULL);
        fr = (uintptr_t)alloc_buf;

        fr = (fr + alignment - 1) / alignment * alignment;

        if (fr == (uintptr_t)alloc_buf)
                return (alloc_buf);

        if ((fr - (uintptr_t)alloc_buf) <= HEADSZ) {
                /*
                 * we hit an edge case, where the space ahead of aligned
                 * address is not sufficient to hold 'header' and hence we
                 * can't free it. So double the allocation request.
                 */
                realloc++;
                free(alloc_buf);
                alloc_size = size + alignment*2;
                if (alloc_size < size) {
                        return (NULL);
                }

                (void) mutex_lock(&mlock);
                alloc_buf = malloc_unlocked(alloc_size, 1);
                (void) mutex_unlock(&mlock);

                if (alloc_buf == NULL)
                        return (NULL);
                fr = (uintptr_t)alloc_buf;

                fr = (fr + alignment - 1) / alignment * alignment;
                if (fr == (uintptr_t)alloc_buf)
                        return (alloc_buf);
                if ((fr - (uintptr_t)alloc_buf) <= HEADSZ) {
                        fr = fr + alignment;
                }
        }

        /*
         *      +-------+               +-------+
         *  +---| <a>   |               | <a>   |--+
         *  |   +-------+<--alloc_buf-->+-------+  |
         *  |   |       |               |       |  |
         *  |   |       |               |       |  |
         *  |   |       |               |       |  |
         *  |   |       |        hd-->  +-------+  |
         *  |   |       |           +---|  <b>  |<-+
         *  |   |       |           |   +-------+<--- fr
         *  |   |       |           |   |       |
         *  |   |       |           |   |       |
         *  |   |       |           |   |       |
         *  |   |       |           |   |       |
         *  |   |       |           |   |       |
         *  |   |       |           |   |       |
         *  |   +-------+           |   +-------+
         *  +-->|  next |           +-->|  next |
         *      +-------+               +-------+
         *
         */
        hd = (struct header *)((char *)fr - minhead);
        (void) mutex_lock(&mlock);
        hd->nextblk = ((struct header *)((char *)alloc_buf - minhead))->nextblk;
        ((struct header *)((char *)alloc_buf - minhead))->nextblk = SETBUSY(hd);
        (void) mutex_unlock(&mlock);
        free(alloc_buf);
        CHECKQ
        return ((void *)fr);
}

void *
valloc(size_t size)
{
        static unsigned pagesize;
        if (size == 0)
                return (NULL);

        if (!pagesize)
                pagesize = sysconf(_SC_PAGESIZE);

        return (memalign(pagesize, size));
}

/*
 * malloc_unlocked(nbytes, nosmall) - Do the real work for malloc
 */

static void *
malloc_unlocked(size_t nbytes, int nosmall)
{
        struct header *blk;
        size_t nb;      /* size of entire block we need */

        /* on first call, initialize */
        if (freeptr[0].nextfree == GROUND) {
                /* initialize arena */
                arena[1].nextblk = (struct header *)BUSY;
                arena[0].nextblk = (struct header *)BUSY;
                lastblk = arenaend = &(arena[1]);
                /* initialize free queue */
                freeptr[0].nextfree = &(freeptr[1]);
                freeptr[1].nextblk = &(arena[0]);
```

```
                freeptr[1].prevfree = &(freeptr[0]);
                /* mark that small blocks not init yet */
        }
        if (nbytes == 0)
                return (NULL);

        if (nbytes <= maxfast && !nosmall) {
                /*
                 * We can allocate out of a holding block
                 */
                struct holdblk *holdblk; /* head of right sized queue */
                struct lblk *lblk;       /* pointer to a little block */
                struct holdblk *newhold;

                if (!change) {
                        int i;
                        /*
                         * This allocates space for hold block
                         * pointers by calling malloc recursively.
                         * Maxfast is temporarily set to 0, to
                         * avoid infinite recursion.  allocate
                         * space for an extra ptr so that an index
                         * is just ->blksz/grain, with the first
                         * ptr unused.
                         */
                        change = 1;     /* change to algorithm params */
                                        /* no longer allowed */
                        /*
                         * temporarily alter maxfast, to avoid
                         * infinite recursion
                         */
                        maxfast = 0;
                        holdhead = (struct holdblk **)
                            malloc_unlocked(sizeof (struct holdblk *) *
                            (fastct + 1), 0);
                        if (holdhead == NULL)
                                return (malloc_unlocked(nbytes, 0));
                        for (i = 1; i <= fastct; i++) {
                                holdhead[i] = HGROUND;
                        }
                        maxfast = fastct * grain;
                }
                /*
                 * Note that this uses the absolute min header size (MINHEAD)
                 * unlike the large block case which uses minhead
                 *
                 * round up to nearest multiple of grain
                 * code assumes grain is a multiple of MINHEAD
                 */
                /* round up to grain */
                nb = (nbytes + grain - 1) / grain * grain;
                holdblk = holdhead[nb / grain];
                nb = nb + MINHEAD;
                /*
                 * look for space in the holding block.  Blocks with
                 * space will be in front of those without
                 */
                if ((holdblk != HGROUND) && (holdblk->lfreeq != LGROUND))  {
                        /* there is space */
                        lblk = holdblk->lfreeq;

                        /*
                         * Now make lfreeq point to a free block.
                         * If lblk has been previously allocated and
                         * freed, it has a valid pointer to use.
                         * Otherwise, lblk is at the beginning of
                         * the unallocated blocks at the end of
                         * the holding block, so, if there is room, take
                         * the next space.  If not, mark holdblk full,
                         * and move holdblk to the end of the queue
                         */
                        if (lblk < holdblk->unused) {
                                /* move to next holdblk, if this one full */
                                if ((holdblk->lfreeq =
                                    CLRSMAL(lblk->header.nextfree)) ==
                                    LGROUND) {
                                        holdhead[(nb-MINHEAD) / grain] =
                                            holdblk->nexthblk;
                                }
                        } else if (((char *)holdblk->unused + nb) <
                            ((char *)holdblk + HOLDSZ(nb))) {
                                holdblk->unused = (struct lblk *)
                                    ((char *)holdblk->unused+nb);
                                holdblk->lfreeq = holdblk->unused;
                        } else {
                                holdblk->unused = (struct lblk *)
                                    ((char *)holdblk->unused+nb);
                                holdblk->lfreeq = LGROUND;
                                holdhead[(nb-MINHEAD)/grain] =
                                    holdblk->nexthblk;
                        }
                        /* mark as busy and small */
                        lblk->header.holder = (struct holdblk *)SETALL(holdblk);
                } else {
                        /* we need a new holding block */
                        newhold = (struct holdblk *)
                            malloc_unlocked(HOLDSZ(nb), 0);
                        if ((char *)newhold == NULL) {
                                return (NULL);
                        }
                        /* add to head of free queue */
                        if (holdblk != HGROUND) {
                                newhold->nexthblk = holdblk;
                                newhold->prevhblk = holdblk->prevhblk;
                                holdblk->prevhblk = newhold;
                                newhold->prevhblk->nexthblk = newhold;
                        } else {
                                newhold->nexthblk = newhold->prevhblk = newhold;
                        }
                        holdhead[(nb-MINHEAD)/grain] = newhold;
                        /* set up newhold */
                        lblk = (struct lblk *)(newhold->space);
                        newhold->lfreeq = newhold->unused =
                            (struct lblk *)((char *)newhold->space+nb);
                        lblk->header.holder = (struct holdblk *)SETALL(newhold);
                        newhold->blksz = nb-MINHEAD;
                }
#ifdef debug
                assert(((struct holdblk *)CLRALL(lblk->header.holder))->blksz >=
                    nbytes);
#endif /* debug */
                return ((char *)lblk + MINHEAD);
        } else {
                /*
                 * We need an ordinary block
                 */
                struct header *newblk;  /* used for creating a block */

                /* get number of bytes we need */
                nb = nbytes + minhead;
                nb = (nb + ALIGNSZ - 1) / ALIGNSZ * ALIGNSZ;    /* align */
                nb = (nb > MINBLKSZ) ? nb : MINBLKSZ;
```

```
                /*
                 * see if there is a big enough block
                 * If none exists, you will get to freeptr[1].
                 * freeptr[1].next = &arena[0], so when you do the test,
                 * the result is a large positive number, since arena[0]
                 * comes before all blocks.  Arena[0] is marked busy so
                 * that it will not be compacted.  This kludge is for the
                 * sake of the almighty efficiency.
                 */
                /* check that a very large request won't cause an inf. loop */

                if ((freeptr[1].nextblk-&(freeptr[1])) < nb) {
                        return (NULL);
                } else {
                        struct header *next;            /* following block */
                        struct header *nextnext;        /* block after next */

                        blk = freeptr;
                        do {
                                blk = blk->nextfree;
                                /* see if we can compact */
                                next = blk->nextblk;
                                if (!TESTBUSY(nextnext = next->nextblk)) {
                                        do {
                                                DELFREEQ(next);
                                                next = nextnext;
                                                nextnext = next->nextblk;
                                        } while (!TESTBUSY(nextnext));
                                        /*
                                         * next will be at most == to lastblk,
                                         * but I think the >= test is faster
                                         */
                                        if (next >= arenaend)
                                                lastblk = blk;
                                        blk->nextblk = next;
                                }
                        } while (((char *)(next) - (char *)blk) < nb);
                }
                /*
                 * if we didn't find a block, get more memory
                 */
                if (blk == &(freeptr[1])) {
                        /*
                         * careful coding could likely replace
                         * newend with arenaend
                         */
                        struct header *newend;  /* new end of arena */
                        ssize_t nget;   /* number of words to get */

                        /*
                         * Three cases - 1. There is space between arenaend
                         *                  and the break value that will become
                         *                  a permanently allocated block.
                         *               2. Case 1 is not true, and the last
                         *                  block is allocated.
                         *               3. Case 1 is not true, and the last
                         *                  block is free
                         */
                        if ((newblk = (struct header *)sbrk(0)) !=
                            (struct header *)((char *)arenaend + HEADSZ)) {
                                /* case 1 */
#ifdef debug
                                if (case1count++ > 0)
                                    (void) write(2, "Case 1 hit more that once."
                                        " brk or sbrk?\n", 41);
#endif
```

```
                                /* get size to fetch */
                                nget = nb + HEADSZ;
                                /* round up to a block */
                                nget = (nget + BLOCKSZ - 1)/BLOCKSZ * BLOCKSZ;
                                assert((uintptr_t)newblk % ALIGNSZ == 0);
                                /* get memory */
                                if (morecore(nget) == (void *)-1)
                                        return (NULL);
                                /* add to arena */
                                newend = (struct header *)((char *)newblk + nget
                                    - HEADSZ);
                                assert((uintptr_t)newblk % ALIGNSZ == 0);
                                newend->nextblk = SETBUSY(&(arena[1]));
/* ???  newblk ?? */
                                newblk->nextblk = newend;

                                /*
                                 * space becomes a permanently allocated block.
                                 * This is likely not mt-safe as lock is not
                                 * shared with brk or sbrk
                                 */
                                arenaend->nextblk = SETBUSY(newblk);
                                /* adjust other pointers */
                                arenaend = newend;
                                lastblk = newblk;
                                blk = newblk;
                        } else if (TESTBUSY(lastblk->nextblk)) {
                                /* case 2 */
                                nget = (nb + BLOCKSZ - 1) / BLOCKSZ * BLOCKSZ;
                                if (morecore(nget) == (void *)-1)
                                        return (NULL);
                                /* block must be word aligned */
                                assert(((uintptr_t)newblk%ALIGNSZ) == 0);
                                /*
                                 * stub at old arenaend becomes first word
                                 * in blk
                                 */
/* ???          newblk = arenaend; */

                                newend =
                                    (struct header *)((char *)arenaend+nget);
                                newend->nextblk = SETBUSY(&(arena[1]));
                                arenaend->nextblk = newend;
                                lastblk = blk = arenaend;
                                arenaend = newend;
                        } else {
                                /* case 3 */
                                /*
                                 * last block in arena is at end of memory and
                                 * is free
                                 */
                                /* 1.7 had this backward without cast */
                                nget = nb -
                                    ((char *)arenaend - (char *)lastblk);
                                nget = (nget + (BLOCKSZ - 1)) /
                                    BLOCKSZ * BLOCKSZ;
                                assert(((uintptr_t)newblk % ALIGNSZ) == 0);
                                if (morecore(nget) == (void *)-1)
                                        return (NULL);
                                /* combine with last block, put in arena */
                                newend = (struct header *)
                                    ((char *)arenaend + nget);
                                arenaend = lastblk->nextblk = newend;
                                newend->nextblk = SETBUSY(&(arena[1]));
                                /* set which block to use */
                                blk = lastblk;
```

```
                                DELFREEQ(blk);
                        }
                } else {
                        struct header *nblk;    /* next block */

                        /* take block found of free queue */
                        DELFREEQ(blk);
                        /*
                         * make head of free queue immediately follow blk,
                         * unless blk was at the end of the queue
                         */
                        nblk = blk->nextfree;
                        if (nblk != &(freeptr[1])) {
                                MOVEHEAD(nblk);
                        }
                }
                /* blk now points to an adequate block */
                if (((char *)blk->nextblk - (char *)blk) - nb >= MINBLKSZ) {
                        /* carve out the right size block */
                        /* newblk will be the remainder */
                        newblk = (struct header *)((char *)blk + nb);
                        newblk->nextblk = blk->nextblk;
                        /* mark the block busy */
                        blk->nextblk = SETBUSY(newblk);
                        ADDFREEQ(newblk);
                        /* if blk was lastblk, make newblk lastblk */
                        if (blk == lastblk)
                                lastblk = newblk;
                } else {
                        /* just mark the block busy */
                        blk->nextblk = SETBUSY(blk->nextblk);
                }
        }
        CHECKQ
        assert((char *)CLRALL(blk->nextblk) -
            ((char *)blk + minhead) >= nbytes);
        assert((char *)CLRALL(blk->nextblk) -
            ((char *)blk + minhead) < nbytes + MINBLKSZ);
        return ((char *)blk + minhead);
}

/*
 * free(ptr) - free block that user thinks starts at ptr
 *
 *      input - ptr-1 contains the block header.
 *              If the header points forward, we have a normal
 *                      block pointing to the next block
 *              if the header points backward, we have a small
 *                      block from a holding block.
 *              In both cases, the busy bit must be set
 */

void
free(void *ptr)
{
        (void) mutex_lock(&mlock);
        free_unlocked(ptr);
        (void) mutex_unlock(&mlock);
}

/*
 * free_unlocked(ptr) - Do the real work for free()
 */

void
free_unlocked(void *ptr)
```

```
{
        struct holdblk *holdblk;        /* block holding blk */
        struct holdblk *oldhead;        /* former head of the hold block */
                                        /* queue containing blk's holder */

        if (ptr == NULL)
                return;
        if (TESTSMAL(((struct header *)((char *)ptr - MINHEAD))->nextblk)) {
                struct lblk     *lblk;  /* pointer to freed block */
                ssize_t         offset; /* choice of header lists */

                lblk = (struct lblk *)CLRBUSY((char *)ptr - MINHEAD);
                assert((struct header *)lblk < arenaend);
                assert((struct header *)lblk > arena);
                /* allow twits (e.g. awk) to free a block twice */
                holdblk = lblk->header.holder;
                if (!TESTBUSY(holdblk))
                        return;
                holdblk = (struct holdblk *)CLRALL(holdblk);
                /* put lblk on its hold block's free list */
                lblk->header.nextfree = SETSMAL(holdblk->lfreeq);
                holdblk->lfreeq = lblk;
                /* move holdblk to head of queue, if its not already there */
                offset = holdblk->blksz / grain;
                oldhead = holdhead[offset];
                if (oldhead != holdblk) {
                        /* first take out of current spot */
                        holdhead[offset] = holdblk;
                        holdblk->nexthblk->prevhblk = holdblk->prevhblk;
                        holdblk->prevhblk->nexthblk = holdblk->nexthblk;
                        /* now add at front */
                        holdblk->nexthblk = oldhead;
                        holdblk->prevhblk = oldhead->prevhblk;
                        oldhead->prevhblk = holdblk;
                        holdblk->prevhblk->nexthblk = holdblk;
                }
        } else {
                struct header *blk;     /* real start of block */
                struct header *next;    /* next = blk->nextblk */
                struct header *nextnext;        /* block after next */

                blk = (struct header *)((char *)ptr - minhead);
                next = blk->nextblk;
                /* take care of twits (e.g. awk) who return blocks twice */
                if (!TESTBUSY(next))
                        return;
                blk->nextblk = next = CLRBUSY(next);
                ADDFREEQ(blk);
                /* see if we can compact */
                if (!TESTBUSY(nextnext = next->nextblk)) {
                        do {
                                DELFREEQ(next);
                                next = nextnext;
                        } while (!TESTBUSY(nextnext = next->nextblk));
                        if (next == arenaend) lastblk = blk;
                        blk->nextblk = next;
                }
        }
        CHECKQ
}


/*
 * realloc(ptr, size) - give the user a block of size "size", with
 *                          the contents pointed to by ptr.  Free ptr.
 */
```

```
void *
realloc(void *ptr, size_t size)
{
        void    *retval;

        (void) mutex_lock(&mlock);
        retval = realloc_unlocked(ptr, size);
        (void) mutex_unlock(&mlock);
        return (retval);
}


/*
 * realloc_unlocked(ptr) - Do the real work for realloc()
 */

static void *
realloc_unlocked(void *ptr, size_t size)
{
        struct header *blk;     /* block ptr is contained in */
        size_t trusize; /* block size as allocater sees it */
        char *newptr;                   /* pointer to user's new block */
        size_t cpysize; /* amount to copy */
        struct header *next;    /* block after blk */

        if (ptr == NULL)
                return (malloc_unlocked(size, 0));

        if (size == 0) {
                free_unlocked(ptr);
                return (NULL);
        }

        if (TESTSMAL(((struct lblk *)((char *)ptr - MINHEAD))->
            header.holder)) {
                /*
                 * we have a special small block which can't be expanded
                 *
                 * This makes the assumption that even if the user is
                 * reallocating a free block, malloc doesn't alter the contents
                 * of small blocks
                 */
                newptr = malloc_unlocked(size, 0);
                if (newptr == NULL)
                        return (NULL);
                /* this isn't to save time--its to protect the twits */
                if ((char *)ptr != newptr) {
                        struct lblk *lblk;
                        lblk = (struct lblk *)((char *)ptr - MINHEAD);
                        cpysize = ((struct holdblk *)
                            CLRALL(lblk->header.holder))->blksz;
                        cpysize = (size > cpysize) ? cpysize : size;
                        (void) memcpy(newptr, ptr, cpysize);
                        free_unlocked(ptr);
                }
        } else {
                blk = (struct header *)((char *)ptr - minhead);
                next = blk->nextblk;
                /*
                 * deal with twits who reallocate free blocks
                 *
                 * if they haven't reset minblk via getopt, that's
                 * their problem
                 */
                if (!TESTBUSY(next)) {
                        DELFREEQ(blk);
                        blk->nextblk = SETBUSY(next);
                }
                next = CLRBUSY(next);
                /* make blk as big as possible */
                if (!TESTBUSY(next->nextblk)) {
                        do {
                                DELFREEQ(next);
                                next = next->nextblk;
                        } while (!TESTBUSY(next->nextblk));
                        blk->nextblk = SETBUSY(next);
                        if (next >= arenaend) lastblk = blk;
                }
                /* get size we really need */
                trusize = size+minhead;
                trusize = (trusize + ALIGNSZ - 1)/ALIGNSZ*ALIGNSZ;
                trusize = (trusize >= MINBLKSZ) ? trusize : MINBLKSZ;
                /* see if we have enough */
                /* this isn't really the copy size, but I need a register */
                cpysize = (char *)next - (char *)blk;
                if (cpysize >= trusize) {
                        /* carve out the size we need */
                        struct header *newblk;  /* remainder */

                        if (cpysize - trusize >= MINBLKSZ) {
                                /*
                                 * carve out the right size block
                                 * newblk will be the remainder
                                 */
                                newblk = (struct header *)((char *)blk +
                                    trusize);
                                newblk->nextblk = next;
                                blk->nextblk = SETBUSY(newblk);
                                /* at this point, next is invalid */
                                ADDFREEQ(newblk);
                                /* if blk was lastblk, make newblk lastblk */
                                if (blk == lastblk)
                                        lastblk = newblk;
                        }
                        newptr = ptr;
                } else {
                        /* bite the bullet, and call malloc */
                        cpysize = (size > cpysize) ? cpysize : size;
                        newptr = malloc_unlocked(size, 0);
                        if (newptr == NULL)
                                return (NULL);
                        (void) memcpy(newptr, ptr, cpysize);
                        free_unlocked(ptr);
                }
        }
        return (newptr);
}


/*
 * calloc - allocate and clear memory block
 */

void *
calloc(size_t num, size_t size)
{
        char *mp;
        size_t total;

        if (num == 0 || size == 0) {
                total = 0;
```

```
        } else {
                total = num * size;

                /* check for overflow */
                if ((total / num) != size) {
                        errno = ENOMEM;
                        return (0);
                }
        }
#endif /* ! codereview */

        mp = malloc(total);
        num *= size;
        mp = malloc(num);
        if (mp == NULL)
                return (NULL);
        (void) memset(mp, 0, total);
        (void) memset(mp, 0, num);
        return (mp);
}
```

______unchanged_portion_omitted_

```
**********************************************************
    1459 Wed Jun 11 19:45:19 2014
new/usr/src/lib/libmapmalloc/common/calloc.c
XXXX all calloc() implementations should check overflow
**********************************************************
/*
 * 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 2008 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 */

#pragma ident   "%Z%%M% %I%     %E% SMI"


#include <stdlib.h>
#include <string.h>
#include <errno.h>

#endif /* ! codereview */
/*
 * calloc - allocate and clear memory block
 */

void *
calloc(size_t num, size_t size)
{
        void *mp;
        size_t total;

        if (num == 0 || size == 0) {
                total = 0;
        } else {
                total = num * size;

                /* check for overflow */
                if (total / num != size) {
                        errno = ENOMEM;
                        return (NULL);
                }
        }
#endif /* ! codereview */

        mp = malloc(total);
        num *= size;
        mp = malloc(num);
        if (mp == NULL)
                return (NULL);
```

```
        (void) memset(mp, 0, total);
        (void) memset(mp, 0, num);
        return (mp);
}
______unchanged_portion_omitted_
```

****
   42752 Wed Jun 11 19:45:20 2014
new/usr/src/lib/libmtmalloc/common/mtmalloc.c
XXXX all calloc() implementations should check overflow
****

______unchanged_portion_omitted_

```
void *
calloc(size_t nelem, size_t bytes)
{
        void * ptr;
        size_t size;

        if (nelem == 0 || bytes == 0) {
                size = 0;
        } else {
                size = nelem * bytes;

                /* check for overflow */
                if (size / nelem != bytes) {
                        errno = ENOMEM;
                        return (NULL);
                }
        }
        size_t size = nelem * bytes;

        ptr = malloc(size);
        if (ptr == NULL)
                return (NULL);
        (void) memset(ptr, 0, size);

        return (ptr);
}
```

______unchanged_portion_omitted_