****
7550 Mon Jun 16 13:23:23 2014
new/usr/src/cmd/sgs/rtld/common/malloc.c
4922 all calloc() implementations should check for overflow
****
_____***unchanged_portion_omitted_***

```
 206 void *
 207 calloc(size_t num, size_t size)
 208 {
 209         void *  mp;
 210         size_t  total;
 211 #endif /* ! codereview */

 213         if (num == 0 || size == 0) {
 214                 total = 0;
 215         } else {
 216                 total = num * size;

 218                 /* check for overflow */
 219                 if ((total / num) != size) {
 220                         errno = ENOMEM;
 210         num *= size;
 211         if ((mp = malloc(num)) == NULL)
 221                         return (NULL);
 222                 }
 223         }

 225         if ((mp = malloc(total)) == NULL)
 226                 return (NULL);
 227         (void) memset(mp, 0, total);
 213         (void) memset(mp, 0, num);
 228         return (mp);
 229 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   31834 Mon Jun 16 13:23:24 2014
new/usr/src/lib/libmalloc/common/malloc.c
4922 all calloc() implementations should check for 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
```

```
126                                * block to be pointed to by the last block.
127                                */

129 #define freeptr (arena + 2)
130                               /* first and last entry in free list */
131 static struct header *arenaend; /* ptr to block marking high end of arena */
132 static struct header *lastblk;  /* the highest block in the arena */
133 static struct holdblk **holdhead;   /* pointer to array of head pointers */
134                                     /* to holding block chains */
135 /*
136  * In order to save time calculating indices, the array is 1 too
137  * large, and the first element is unused
138  *
139  * Variables controlling algorithm, esp. how holding blocs are used
140  */
141 static int numlblks = NUMLBLKS;
142 static int minhead = MINHEAD;
143 static int change = 0;  /* != 0, once param changes are no longer allowed */
144 static int fastct = FASTCT;
145 static unsigned int maxfast = MAXFAST;
146 /* number of small block sizes to map to one size */

148 static int grain = ALIGNSZ;

150 #ifdef debug
151 static int case1count = 0;

153 static void
154 checkq(void)
155 {
156         register struct header *p;

158         p = &freeptr[0];

160         /* check forward */
161         /*CSTYLED*/
162         while (p != &freeptr[1]) {
163                 p = p->nextfree;
164                 assert(p->prevfree->nextfree == p);
165         }

167         /* check backward */
168         /*CSTYLED*/
169         while (p != &freeptr[0]) {
170                 p = p->prevfree;
171                 assert(p->nextfree->prevfree == p);
172         }
173 }
174 #endif


177 /*
178  * malloc(nbytes) - give a user nbytes to use
179  */

181 void *
182 malloc(size_t nbytes)
183 {
184         void *ret;

186         (void) mutex_lock(&mlock);
187         ret = malloc_unlocked(nbytes, 0);
188         (void) mutex_unlock(&mlock);
189         return (ret);
190 }
```



```
192 /*
193  * Use malloc_unlocked() to get the address to start with; Given this
194  * address, find out the closest address that aligns with the request
195  * and return that address after doing some house keeping (refer to the
196  * ascii art below).
197  */
198 void *
199 memalign(size_t alignment, size_t size)
200 {
201         void *alloc_buf;
202         struct header *hd;
203         size_t alloc_size;
204         uintptr_t fr;
205         static int realloc;

207         if (size == 0 || alignment == 0 ||
208             (alignment & (alignment - 1)) != 0) {
209                 return (NULL);
210         }
211         if (alignment <= ALIGNSZ)
212                 return (malloc(size));

214         alloc_size = size + alignment;
215         if (alloc_size < size) { /* overflow */
216                 return (NULL);
217         }

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

223         if (alloc_buf == NULL)
224                 return (NULL);
225         fr = (uintptr_t)alloc_buf;

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

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

232         if ((fr - (uintptr_t)alloc_buf) <= HEADSZ) {
233                 /*
234                  * we hit an edge case, where the space ahead of aligned
235                  * address is not sufficient to hold 'header' and hence we
236                  * can't free it. So double the allocation request.
237                  */
238                 realloc++;
239                 free(alloc_buf);
240                 alloc_size = size + alignment*2;
241                 if (alloc_size < size) {
242                         return (NULL);
243                 }

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

249                 if (alloc_buf == NULL)
250                         return (NULL);
251                 fr = (uintptr_t)alloc_buf;

253                 fr = (fr + alignment - 1) / alignment * alignment;
254                 if (fr == (uintptr_t)alloc_buf)
255                         return (alloc_buf);
256                 if ((fr - (uintptr_t)alloc_buf) <= HEADSZ) {
257                         fr = fr + alignment;
```

```
258                 }
259         }

261         /*
262          *      +-------+               +-------+
263          *  +---| <a>   |               | <a>   |--+
264          *  |   +-------+<--alloc_buf-->+-------+  |
265          *  |   |       |               |       |  |
266          *  |   |       |               |       |  |
267          *  |   |       |               |       |  |
268          *  |   |       |        hd-->  +-------+  |
269          *  |   |       |           +---|  <b>  |<-+
270          *  |   |       |           |   +-------+<--- fr
271          *  |   |       |           |   |       |
272          *  |   |       |           |   |       |
273          *  |   |       |           |   |       |
274          *  |   |       |           |   |       |
275          *  |   |       |           |   |       |
276          *  |   |       |           |   |       |
277          *  |   +-------+           |   +-------+
278          *  +-->|  next |           +-->|  next |
279          *      +-------+               +-------+
280          *
281          */
282         hd = (struct header *)((char *)fr - minhead);
283         (void) mutex_lock(&mlock);
284         hd->nextblk = ((struct header *)((char *)alloc_buf - minhead))->nextblk;
285         ((struct header *)((char *)alloc_buf - minhead))->nextblk = SETBUSY(hd);
286         (void) mutex_unlock(&mlock);
287         free(alloc_buf);
288         CHECKQ
289         return ((void *)fr);
290 }

292 void *
293 valloc(size_t size)
294 {
295         static unsigned pagesize;
296         if (size == 0)
297                 return (NULL);

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

302         return (memalign(pagesize, size));
303 }

305 /*
306  * malloc_unlocked(nbytes, nosmall) - Do the real work for malloc
307  */

309 static void *
310 malloc_unlocked(size_t nbytes, int nosmall)
311 {
312         struct header *blk;
313         size_t nb;      /* size of entire block we need */

315         /* on first call, initialize */
316         if (freeptr[0].nextfree == GROUND) {
317                 /* initialize arena */
318                 arena[1].nextblk = (struct header *)BUSY;
319                 arena[0].nextblk = (struct header *)BUSY;
320                 lastblk = arenaend = &(arena[1]);
321                 /* initialize free queue */
322                 freeptr[0].nextfree = &(freeptr[1]);
323                 freeptr[1].nextblk = &(arena[0]);
```



```
324                 freeptr[1].prevfree = &(freeptr[0]);
325                 /* mark that small blocks not init yet */
326         }
327         if (nbytes == 0)
328                 return (NULL);

330         if (nbytes <= maxfast && !nosmall) {
331                 /*
332                  * We can allocate out of a holding block
333                  */
334                 struct holdblk *holdblk; /* head of right sized queue */
335                 struct lblk *lblk;       /* pointer to a little block */
336                 struct holdblk *newhold;

338                 if (!change) {
339                         int i;
340                         /*
341                          * This allocates space for hold block
342                          * pointers by calling malloc recursively.
343                          * Maxfast is temporarily set to 0, to
344                          * avoid infinite recursion.  allocate
345                          * space for an extra ptr so that an index
346                          * is just ->blksz/grain, with the first
347                          * ptr unused.
348                          */
349                         change = 1;     /* change to algorithm params */
350                                         /* no longer allowed */
351                         /*
352                          * temporarily alter maxfast, to avoid
353                          * infinite recursion
354                          */
355                         maxfast = 0;
356                         holdhead = (struct holdblk **)
357                             malloc_unlocked(sizeof (struct holdblk *) *
358                             (fastct + 1), 0);
359                         if (holdhead == NULL)
360                                 return (malloc_unlocked(nbytes, 0));
361                         for (i = 1; i <= fastct; i++) {
362                                 holdhead[i] = HGROUND;
363                         }
364                         maxfast = fastct * grain;
365                 }
366                 /*
367                  * Note that this uses the absolute min header size (MINHEAD)
368                  * unlike the large block case which uses minhead
369                  *
370                  * round up to nearest multiple of grain
371                  * code assumes grain is a multiple of MINHEAD
372                  */
373                 /* round up to grain */
374                 nb = (nbytes + grain - 1) / grain * grain;
375                 holdblk = holdhead[nb / grain];
376                 nb = nb + MINHEAD;
377                 /*
378                  * look for space in the holding block.  Blocks with
379                  * space will be in front of those without
380                  */
381                 if ((holdblk != HGROUND) && (holdblk->lfreeq != LGROUND))  {
382                         /* there is space */
383                         lblk = holdblk->lfreeq;

385                         /*
386                          * Now make lfreeq point to a free block.
387                          * If lblk has been previously allocated and
388                          * freed, it has a valid pointer to use.
389                          * Otherwise, lblk is at the beginning of
```

```
390                          * the unallocated blocks at the end of
391                          * the holding block, so, if there is room, take
392                          * the next space.  If not, mark holdblk full,
393                          * and move holdblk to the end of the queue
394                          */
395                         if (lblk < holdblk->unused) {
396                                 /* move to next holdblk, if this one full */
397                                 if ((holdblk->lfreeq =
398                                     CLRSMAL(lblk->header.nextfree)) ==
399                                     LGROUND) {
400                                         holdhead[(nb-MINHEAD) / grain] =
401                                             holdblk->nexthblk;
402                                 }
403                         } else if (((char *)holdblk->unused + nb) <
404                             ((char *)holdblk + HOLDSZ(nb))) {
405                                 holdblk->unused = (struct lblk *)
406                                     ((char *)holdblk->unused+nb);
407                                 holdblk->lfreeq = holdblk->unused;
408                         } else {
409                                 holdblk->unused = (struct lblk *)
410                                     ((char *)holdblk->unused+nb);
411                                 holdblk->lfreeq = LGROUND;
412                                 holdhead[(nb-MINHEAD)/grain] =
413                                     holdblk->nexthblk;
414                         }
415                         /* mark as busy and small */
416                         lblk->header.holder = (struct holdblk *)SETALL(holdblk);
417                 } else {
418                         /* we need a new holding block */
419                         newhold = (struct holdblk *)
420                             malloc_unlocked(HOLDSZ(nb), 0);
421                         if ((char *)newhold == NULL) {
422                                 return (NULL);
423                         }
424                         /* add to head of free queue */
425                         if (holdblk != HGROUND) {
426                                 newhold->nexthblk = holdblk;
427                                 newhold->prevhblk = holdblk->prevhblk;
428                                 holdblk->prevhblk = newhold;
429                                 newhold->prevhblk->nexthblk = newhold;
430                         } else {
431                                 newhold->nexthblk = newhold->prevhblk = newhold;
432                         }
433                         holdhead[(nb-MINHEAD)/grain] = newhold;
434                         /* set up newhold */
435                         lblk = (struct lblk *)(newhold->space);
436                         newhold->lfreeq = newhold->unused =
437                             (struct lblk *)((char *)newhold->space+nb);
438                         lblk->header.holder = (struct holdblk *)SETALL(newhold);
439                         newhold->blksz = nb-MINHEAD;
440                 }
441 #ifdef debug
442                 assert(((struct holdblk *)CLRALL(lblk->header.holder))->blksz >=
443                     nbytes);
444 #endif /* debug */
445                 return ((char *)lblk + MINHEAD);
446         } else {
447                 /*
448                  * We need an ordinary block
449                  */
450                 struct header *newblk;  /* used for creating a block */

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



```
456                 /*
457                  * see if there is a big enough block
458                  * If none exists, you will get to freeptr[1].
459                  * freeptr[1].next = &arena[0], so when you do the test,
460                  * the result is a large positive number, since arena[0]
461                  * comes before all blocks.  Arena[0] is marked busy so
462                  * that it will not be compacted.  This kludge is for the
463                  * sake of the almighty efficiency.
464                  */
465                 /* check that a very large request won't cause an inf. loop */

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

473                         blk = freeptr;
474                         do {
475                                 blk = blk->nextfree;
476                                 /* see if we can compact */
477                                 next = blk->nextblk;
478                                 if (!TESTBUSY(nextnext = next->nextblk)) {
479                                         do {
480                                                 DELFREEQ(next);
481                                                 next = nextnext;
482                                                 nextnext = next->nextblk;
483                                         } while (!TESTBUSY(nextnext));
484                                         /*
485                                          * next will be at most == to lastblk,
486                                          * but I think the >= test is faster
487                                          */
488                                         if (next >= arenaend)
489                                                 lastblk = blk;
490                                         blk->nextblk = next;
491                                 }
492                         } while (((char *)(next) - (char *)blk) < nb);
493                 }
494                 /*
495                  * if we didn't find a block, get more memory
496                  */
497                 if (blk == &(freeptr[1])) {
498                         /*
499                          * careful coding could likely replace
500                          * newend with arenaend
501                          */
502                         struct header *newend;  /* new end of arena */
503                         ssize_t nget;   /* number of words to get */

505                         /*
506                          * Three cases - 1. There is space between arenaend
507                          *                  and the break value that will become
508                          *                  a permanently allocated block.
509                          *               2. Case 1 is not true, and the last
510                          *                  block is allocated.
511                          *               3. Case 1 is not true, and the last
512                          *                  block is free
513                          */
514                         if ((newblk = (struct header *)sbrk(0)) !=
515                             (struct header *)((char *)arenaend + HEADSZ)) {
516                                 /* case 1 */
517 #ifdef debug
518                                 if (case1count++ > 0)
519                                     (void) write(2, "Case 1 hit more that once."
520                                         " brk or sbrk?\n", 41);
521 #endif
```

```
522                                 /* get size to fetch */
523                                 nget = nb + HEADSZ;
524                                 /* round up to a block */
525                                 nget = (nget + BLOCKSZ - 1)/BLOCKSZ * BLOCKSZ;
526                                 assert((uintptr_t)newblk % ALIGNSZ == 0);
527                                 /* get memory */
528                                 if (morecore(nget) == (void *)-1)
529                                         return (NULL);
530                                 /* add to arena */
531                                 newend = (struct header *)((char *)newblk + nget
532                                     - HEADSZ);
533                                 assert((uintptr_t)newblk % ALIGNSZ == 0);
534                                 newend->nextblk = SETBUSY(&(arena[1]));
535 /* ???  newblk ?? */
536                                 newblk->nextblk = newend;
537
538                                 /*
539                                  * space becomes a permanently allocated block.
540                                  * This is likely not mt-safe as lock is not
541                                  * shared with brk or sbrk
542                                  */
543                                 arenaend->nextblk = SETBUSY(newblk);
544                                 /* adjust other pointers */
545                                 arenaend = newend;
546                                 lastblk = newblk;
547                                 blk = newblk;
548                         } else if (TESTBUSY(lastblk->nextblk)) {
549                                 /* case 2 */
550                                 nget = (nb + BLOCKSZ - 1) / BLOCKSZ * BLOCKSZ;
551                                 if (morecore(nget) == (void *)-1)
552                                         return (NULL);
553                                 /* block must be word aligned */
554                                 assert(((uintptr_t)newblk%ALIGNSZ) == 0);
555                                 /*
556                                  * stub at old arenaend becomes first word
557                                  * in blk
558                                  */
559 /* ???          newblk = arenaend; */
560
561                                 newend =
562                                     (struct header *)((char *)arenaend+nget);
563                                 newend->nextblk = SETBUSY(&(arena[1]));
564                                 arenaend->nextblk = newend;
565                                 lastblk = blk = arenaend;
566                                 arenaend = newend;
567                         } else {
568                                 /* case 3 */
569                                 /*
570                                  * last block in arena is at end of memory and
571                                  * is free
572                                  */
573                                 /* 1.7 had this backward without cast */
574                                 nget = nb -
575                                     ((char *)arenaend - (char *)lastblk);
576                                 nget = (nget + (BLOCKSZ - 1)) /
577                                     BLOCKSZ * BLOCKSZ;
578                                 assert(((uintptr_t)newblk % ALIGNSZ) == 0);
579                                 if (morecore(nget) == (void *)-1)
580                                         return (NULL);
581                                 /* combine with last block, put in arena */
582                                 newend = (struct header *)
583                                     ((char *)arenaend + nget);
584                                 arenaend = lastblk->nextblk = newend;
585                                 newend->nextblk = SETBUSY(&(arena[1]));
586                                 /* set which block to use */
587                                 blk = lastblk;
```



```
588                                 DELFREEQ(blk);
589                         }
590                 } else {
591                         struct header *nblk;    /* next block */
592
593                         /* take block found of free queue */
594                         DELFREEQ(blk);
595                         /*
596                          * make head of free queue immediately follow blk,
597                          * unless blk was at the end of the queue
598                          */
599                         nblk = blk->nextfree;
600                         if (nblk != &(freeptr[1])) {
601                                 MOVEHEAD(nblk);
602                         }
603                 }
604                 /* blk now points to an adequate block */
605                 if (((char *)blk->nextblk - (char *)blk) - nb >= MINBLKSZ) {
606                         /* carve out the right size block */
607                         /* newblk will be the remainder */
608                         newblk = (struct header *)((char *)blk + nb);
609                         newblk->nextblk = blk->nextblk;
610                         /* mark the block busy */
611                         blk->nextblk = SETBUSY(newblk);
612                         ADDFREEQ(newblk);
613                         /* if blk was lastblk, make newblk lastblk */
614                         if (blk == lastblk)
615                                 lastblk = newblk;
616                 } else {
617                         /* just mark the block busy */
618                         blk->nextblk = SETBUSY(blk->nextblk);
619                 }
620         }
621         CHECKQ
622         assert((char *)CLRALL(blk->nextblk) -
623             ((char *)blk + minhead) >= nbytes);
624         assert((char *)CLRALL(blk->nextblk) -
625             ((char *)blk + minhead) < nbytes + MINBLKSZ);
626         return ((char *)blk + minhead);
627 }
628
629 /*
630  * free(ptr) - free block that user thinks starts at ptr
631  *
632  *      input - ptr-1 contains the block header.
633  *              If the header points forward, we have a normal
634  *                      block pointing to the next block
635  *              if the header points backward, we have a small
636  *                      block from a holding block.
637  *              In both cases, the busy bit must be set
638  */
639
640 void
641 free(void *ptr)
642 {
643         (void) mutex_lock(&mlock);
644         free_unlocked(ptr);
645         (void) mutex_unlock(&mlock);
646 }
647
648 /*
649  * free_unlocked(ptr) - Do the real work for free()
650  */
651
652 void
653 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)) {
```

```
786                         DELFREEQ(blk);
787                         blk->nextblk = SETBUSY(next);
788                 }
789                 next = CLRBUSY(next);
790                 /* make blk as big as possible */
791                 if (!TESTBUSY(next->nextblk)) {
792                         do {
793                                 DELFREEQ(next);
794                                 next = next->nextblk;
795                         } while (!TESTBUSY(next->nextblk));
796                         blk->nextblk = SETBUSY(next);
797                         if (next >= arenaend) lastblk = blk;
798                 }
799                 /* get size we really need */
800                 trusize = size+minhead;
801                 trusize = (trusize + ALIGNSZ - 1)/ALIGNSZ*ALIGNSZ;
802                 trusize = (trusize >= MINBLKSZ) ? trusize : MINBLKSZ;
803                 /* see if we have enough */
804                 /* this isn't really the copy size, but I need a register */
805                 cpysize = (char *)next - (char *)blk;
806                 if (cpysize >= trusize) {
807                         /* carve out the size we need */
808                         struct header *newblk;  /* remainder */

810                         if (cpysize - trusize >= MINBLKSZ) {
811                                 /*
812                                  * carve out the right size block
813                                  * newblk will be the remainder
814                                  */
815                                 newblk = (struct header *)((char *)blk +
816                                     trusize);
817                                 newblk->nextblk = next;
818                                 blk->nextblk = SETBUSY(newblk);
819                                 /* at this point, next is invalid */
820                                 ADDFREEQ(newblk);
821                                 /* if blk was lastblk, make newblk lastblk */
822                                 if (blk == lastblk)
823                                         lastblk = newblk;
824                         }
825                         newptr = ptr;
826                 } else {
827                         /* bite the bullet, and call malloc */
828                         cpysize = (size > cpysize) ? cpysize : size;
829                         newptr = malloc_unlocked(size, 0);
830                         if (newptr == NULL)
831                                 return (NULL);
832                         (void) memcpy(newptr, ptr, cpysize);
833                         free_unlocked(ptr);
834                 }
835         }
836         return (newptr);
837 }


840 /*
841  * calloc - allocate and clear memory block
842  */

844 void *
845 calloc(size_t num, size_t size)
846 {
847         char *mp;
848         size_t total;

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



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

855                 /* check for overflow */
856                 if ((total / num) != size) {
857                         errno = ENOMEM;
858                         return (NULL);
859                 }
860         }
861 #endif /* ! codereview */

863         mp = malloc(total);
 40         num *= size;
 41         mp = malloc(num);
864         if (mp == NULL)
865                 return (NULL);
866         (void) memset(mp, 0, total);
 44         (void) memset(mp, 0, num);
867         return (mp);
868 }
```

_____*unchanged_portion_omitted_*

```
**********************************************************
    1461 Mon Jun 16 13:23:24 2014
new/usr/src/lib/libmapmalloc/common/calloc.c
4922 all calloc() implementations should check for 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_*

****
42754 Mon Jun 16 13:23:25 2014
new/usr/src/lib/libmtmalloc/common/mtmalloc.c
4922 all calloc() implementations should check for overflow
****
_____***unchanged_portion_omitted_***

```
 330 void *
 331 calloc(size_t nelem, size_t bytes)
 332 {
 333         void * ptr;
 334         size_t size;

 336         if (nelem == 0 || bytes == 0) {
 337                 size = 0;
 338         } else {
 339                 size = nelem * bytes;

 341                 /* check for overflow */
 342                 if ((size / nelem) != bytes) {
 343                         errno = ENOMEM;
 344                         return (NULL);
 345                 }
 346         }
 334         size_t size = nelem * bytes;

 348         ptr = malloc(size);
 349         if (ptr == NULL)
 350                 return (NULL);
 351         (void) memset(ptr, 0, size);

 353         return (ptr);
 354 }
```

_____***unchanged_portion_omitted_***