

new/usr/src/cmd/sgs/rtdl/common/malloc.c

1

```
*****  
7548 Wed Jun 11 19:45:18 2014
```

new/usr/src/cmd/sgs/rtdl/common/malloc.c

XXXX all calloc() implementations should check 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_

31831 Wed Jun 11 19:45:19 2014

new/usr/src/lib/libmalloc/common/malloc.c

XXXX all calloc() implementations should check overflow

```

1 /*
2  * CDDL HEADER START
3  *
4  * The contents of this file are subject to the terms of the
5  * Common Development and Distribution License (the "License").
6  * You may not use this file except in compliance with the License.
7  *
8  * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
9  * or http://www.opensolaris.org/os/licensing.
10 * See the License for the specific language governing permissions
11 * and limitations under the License.
12 *
13 * When distributing Covered Code, include this CDDL HEADER in each
14 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
15 * If applicable, add the following below this CDDL HEADER, with the
16 * fields enclosed by brackets "[]" replaced with your own identifying
17 * information: Portions Copyright [yyyy] [name of copyright owner]
18 *
19 * CDDL HEADER END
20 */

22 /*
23  * Copyright 2008 Sun Microsystems, Inc. All rights reserved.
24  * Use is subject to license terms.
25  */

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

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

30 #include <sys/types.h>

32 #ifndef debug
33 #define NDEBUG
34 #endif

36 #include <stdlib.h>
37 #include <string.h>
38 #include <errno.h>
39 #endif /* ! codereview */
40 #include "assert.h"
41 #include "malloc.h"
42 #include "mallint.h"
43 #include <thread.h>
44 #include <pthread.h>
45 #include <synch.h>
46 #include <unistd.h>
47 #include <limits.h>

49 static mutex_t mlock = DEFAULTMUTEX;
50 static ssize_t freespace(struct holdblk *);
51 static void *malloc_unlocked(size_t, int);
52 static void *realloc_unlocked(void *, size_t);
53 static void free_unlocked(void *);
54 static void *morecore(size_t);

56 /*
57  * use level memory allocator (malloc, free, realloc)
58  *
59  * -malloc, free, realloc and mallopt form a memory allocator

```

```

60 * similar to malloc, free, and realloc. The routines
61 * here are much faster than the original, with slightly worse
62 * space usage (a few percent difference on most input). They
63 * do not have the property that data in freed blocks is left
64 * untouched until the space is reallocated.
65 *
66 * -Memory is kept in the "arena", a singly linked list of blocks.
67 * These blocks are of 3 types.
68 * 1. A free block. This is a block not in use by the
69 * user. It has a 3 word header. (See description
70 * of the free queue.)
71 * 2. An allocated block. This is a block the user has
72 * requested. It has only a 1 word header, pointing
73 * to the next block of any sort.
74 * 3. A permanently allocated block. This covers space
75 * aquired by the user directly through sbrk(). It
76 * has a 1 word header, as does 2.
77 * Blocks of type 1 have the lower bit of the pointer to the
78 * nextblock = 0. Blocks of type 2 and 3 have that bit set,
79 * to mark them busy.
80 *
81 * -Unallocated blocks are kept on an unsorted doubly linked
82 * free list.
83 *
84 * -Memory is allocated in blocks, with sizes specified by the
85 * user. A circular first-fit strategy is used, with a roving
86 * head of the free queue, which prevents bunching of small
87 * blocks at the head of the queue.
88 *
89 * -Compaction is performed at free time of any blocks immediately
90 * following the freed block. The freed block will be combined
91 * with a preceding block during the search phase of malloc.
92 * Since a freed block is added at the front of the free queue,
93 * which is moved to the end of the queue if considered and
94 * rejected during the search, fragmentation only occurs if
95 * a block with a contiguous preceding block that is free is
96 * freed and reallocated on the next call to malloc. The
97 * time savings of this strategy is judged to be worth the
98 * occasional waste of memory.
99 *
100 * -Small blocks (of size < MAXSIZE) are not allocated directly.
101 * A large "holding" block is allocated via a recursive call to
102 * malloc. This block contains a header and ?????? small blocks.
103 * Holding blocks for a given size of small block (rounded to the
104 * nearest ALIGNSZ bytes) are kept on a queue with the property that any
105 * holding block with an unused small block is in front of any without.
106 * A list of free blocks is kept within the holding block.
107 */

109 /*
110 * description of arena, free queue, holding blocks etc.
111 *
112 * New compiler and linker does not guarantee order of initialized data.
113 * Define freeptr as arena[2-3] to guarantee it follows arena in memory.
114 * Later code depends on this order.
115 */

117 static struct header arena[4] = {
118     {0, 0, 0},
119     {0, 0, 0},
120     {0, 0, 0},
121     {0, 0, 0}
122 };
123
124 /*
125  * the second word is a minimal block to
126  * 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 caselcount = 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  * +-----+ +-----+
264  * |<a>|<a>|
265  * |-----|<----alloc_buf-----|
266  * |<-----|<-----|
267  * |<-----|<-----|
268  * |<-----|<-----|
269  * |<-----|<-----|
270  * |<-----|<-----|
271  * |<-----|<-----|
272  * |<-----|<-----|
273  * |<-----|<-----|
274  * |<-----|<-----|
275  * |<-----|<-----|
276  * |<-----|<-----|
277  * |<-----|<-----|
278  * |<-----|<-----|
279  * |<-----|<-----|
280  *
281  */
282 hd = (struct header *)((char *)fr - minhead);
283 (void) mutex_lock(&mlck);
284 hd->nextblk = ((struct header *)((char *)alloc_buf - minhead))->nextblk;
285 ((struct header *)((char *)alloc_buf - minhead))->nextblk = SETBUSY(hd);
286 (void) mutex_unlock(&mlck);
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->lfreq != LGROUND)) {
382     /* there is space */
383     lblk = holdblk->lfreq;
385     /*
386      * Now make lfreq 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->lfreeeq =
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->lfreeeq = holdblk->unused;
408     } else {
409         holdblk->unused = (struct lblk *)
410             ((char *)holdblk->unused+nb);
411         holdblk->lfreeeq = 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->lfreeeq = 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 */
451
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 */
466     if ((freeptr[1].nextblk-&(freeptr[1])) < nb) {
467         return (NULL);
468     } else {
469         struct header *next; /* following block */
470         struct header *nextnext; /* block after next */
471
472         blk = freeptr;
473         do {
474             blk = blk->nextfree;
475             /* see if we can compact */
476             next = blk->nextblk;
477             if (!TESTBUSY(nextnext = next->nextblk)) {
478                 do {
479                     DELFREEQ(next);
480                     next = nextnext;
481                     nextnext = next->nextblk;
482                 } while (!TESTBUSY(nextnext));
483                 /*
484                 * next will be at most == to lastblk,
485                 * but I think the >= test is faster
486                 */
487                 if (next >= arenaend)
488                     lastblk = blk;
489                 blk->nextblk = next;
490             }
491         } while (((char *)next) - (char *)blk < nb);
492     }
493     /*
494     * if we didn't find a block, get more memory
495     */
496     if (blk == &(freeptr[1])) {
497         /*
498         * careful coding could likely replace
499         * newend with arenaend
500         */
501         struct header *newend; /* new end of arena */
502         ssize_t nget; /* number of words to get */
503
504         /*
505         * Three cases - 1. There is space between arenaend
506         * and the break value that will become
507         * a permanently allocated block.
508         * 2. Case 1 is not true, and the last
509         * block is allocated.
510         * 3. Case 1 is not true, and the last
511         * block is free
512         */
513         if ((newblk = (struct header *)sbrk(0)) !=
514             (struct header *)((char *)arenaend + HEADSZ)) {
515             /* case 1 */
516             #ifdef debug
517                 if (caselcount++ > 0)
518                     (void) write(2, "Case 1 hit more than once."
519                         " brk or sbrk?\n", 41);
520             #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
537     newblk->nextblk = newend;
538
539     /*
540     * space becomes a permanently allocated block.
541     * This is likely not mt-safe as lock is not
542     * shared with brk or sbrk
543     */
544     arenaend->nextblk = SETBUSY(newblk);
545     /* adjust other pointers */
546     arenaend = newend;
547     lastblk = newblk;
548     blk = newblk;
549     } else if (TESTBUSY(lastblk->nextblk)) {
550     /* case 2 */
551     nget = (nb + BLOCKSZ - 1) / BLOCKSZ * BLOCKSZ;
552     if (morecore(nget) == (void *)-1)
553         return (NULL);
554     /* block must be word aligned */
555     assert(((uintptr_t)newblk%ALIGNSZ) == 0);
556     /*
557     * stub at old arenaend becomes first word
558     * in blk
559     */
560     newblk = arenaend; /*
561
562     newend =
563     (struct header *)((char *)arenaend+nget);
564     newend->nextblk = SETBUSY(&(arena[1]));
565     arenaend->nextblk = newend;
566     lastblk = blk = arenaend;
567     arenaend = newend;
568     } else {
569     /* case 3 */
570     /*
571     * last block in arena is at end of memory and
572     * is free
573     */
574     /* 1.7 had this backward without cast */
575     nget = nb -
576     ((char *)arenaend - (char *)lastblk);
577     nget = (nget + (BLOCKSZ - 1)) /
578     BLOCKSZ * BLOCKSZ;
579     assert(((uintptr_t)newblk % ALIGNSZ) == 0);
580     if (morecore(nget) == (void *)-1)
581         return (NULL);
582     /* combine with last block, put in arena */
583     newend = (struct header *)
584     ((char *)arenaend + nget);
585     arenaend = lastblk->nextblk = newend;
586     newend->nextblk = SETBUSY(&(arena[1]));
587     /* set which block to use */
588     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(&mutex);
644     free_unlocked(ptr);
645     (void) mutex_unlock(&mutex);
646 }
647
648 /*
649 * free_unlocked(ptr) - Do the real work for free()
650 */
651
652 void
653 free_unlocked(void *ptr)

```

```

654 {
655     struct holdblk *holdblk;      /* block holding blk */
656     struct holdblk *oldhead;     /* former head of the hold block */
657     /* queue containing blk's holder */

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

665         lblk = (struct lblk *)CLRBUSY((char *)ptr - MINHEAD);
666         assert((struct header *)lblk < arenaend);
667         assert((struct header *)lblk > arena);
668         /* allow twits (e.g. awk) to free a block twice */
669         holdblk = lblk->header.holder;
670         if (!TESTBUSY(holdblk))
671             return;
672         holdblk = (struct holdblk *)CLRALL(holdblk);
673         /* put lblk on its hold block's free list */
674         lblk->header.nextfree = SETSMAL(holdblk->lfreq);
675         holdblk->lfreq = lblk;
676         /* move holdblk to head of queue, if its not already there */
677         offset = holdblk->blksz / grain;
678         oldhead = holdhead[offset];
679         if (oldhead != holdblk) {
680             /* first take out of current spot */
681             holdhead[offset] = holdblk;
682             holdblk->nexthblk->prevhblk = holdblk->prevhblk;
683             holdblk->prevhblk->nexthblk = holdblk->nexthblk;
684             /* now add at front */
685             holdblk->nexthblk = oldhead;
686             holdblk->prevhblk = oldhead->prevhblk;
687             oldhead->prevhblk = holdblk;
688             holdblk->prevhblk->nexthblk = holdblk;
689         }
690     } else {
691         struct header *blk; /* real start of block */
692         struct header *next; /* next = blk->nextblk */
693         struct header *nextnext; /* block after next */

695         blk = (struct header *)((char *)ptr - minhead);
696         next = blk->nextblk;
697         /* take care of twits (e.g. awk) who return blocks twice */
698         if (!TESTBUSY(next))
699             return;
700         blk->nextblk = next = CLRBUSY(next);
701         ADDFREEQ(blk);
702         /* see if we can compact */
703         if (!TESTBUSY(nextnext = next->nextblk)) {
704             do {
705                 DELFREEQ(next);
706                 next = nextnext;
707             } while (!TESTBUSY(nextnext = next->nextblk));
708             if (next == arenaend) lastblk = blk;
709             blk->nextblk = next;
710         }
711     }
712     CHECKQ
713 }

716 /*
717 * realloc(ptr, size) - give the user a block of size "size", with
718 * the contents pointed to by ptr. Free ptr.
719 */

```

```

721 void *
722 realloc(void *ptr, size_t size)
723 {
724     void *retval;

726     (void) mutex_lock(&lock);
727     retval = realloc_unlocked(ptr, size);
728     (void) mutex_unlock(&lock);
729     return (retval);
730 }

733 /*
734 * realloc_unlocked(ptr) - Do the real work for realloc()
735 */

737 static void *
738 realloc_unlocked(void *ptr, size_t size)
739 {
740     struct header *blk; /* block ptr is contained in */
741     size_t trusize; /* block size as allocator sees it */
742     char *newptr; /* pointer to user's new block */
743     size_t cpysize; /* amount to copy */
744     struct header *next; /* block after blk */

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

749     if (size == 0) {
750         free_unlocked(ptr);
751         return (NULL);
752     }

754     if (TESTSMAL(((struct lblk *)((char *)ptr - MINHEAD))->
755                 header.holder)) {
756         /*
757          * we have a special small block which can't be expanded
758          *
759          * This makes the assumption that even if the user is
760          * reallocating a free block, malloc doesn't alter the contents
761          * of small blocks
762          */
763         newptr = malloc_unlocked(size, 0);
764         if (newptr == NULL)
765             return (NULL);
766         /* this isn't to save time--its to protect the twits */
767         if ((char *)ptr != newptr) {
768             struct lblk *lblk;
769             lblk = (struct lblk *)((char *)ptr - MINHEAD);
770             cpysize = ((struct holdblk *)
771                       CLRALL(lblk->header.holder))->blksz;
772             cpysize = (size > cpysize) ? cpysize : size;
773             (void) memcpy(newptr, ptr, cpysize);
774             free_unlocked(ptr);
775         }
776     } else {
777         blk = (struct header *)((char *)ptr - minhead);
778         next = blk->nextblk;
779         /*
780          * deal with twits who reallocate free blocks
781          *
782          * if they haven't reset minblk via getopt, that's
783          * their problem
784          */
785         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 */
809
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 (0);
859         }
860     }
861 #endif /* ! codereview */

863     mp = malloc(total);
864     num *= size;
865     mp = malloc(num);
866     if (mp == NULL)
867         return (NULL);
868     (void) memset(mp, 0, total);
869     (void) memset(mp, 0, num);
870     return (mp);
871 }

```

_____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
```

```
*****
```

```
1 /*
2  * CDDL HEADER START
3  *
4  * The contents of this file are subject to the terms of the
5  * Common Development and Distribution License (the "License").
6  * You may not use this file except in compliance with the License.
7  *
8  * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
9  * or http://www.opensolaris.org/os/licensing.
10 * See the License for the specific language governing permissions
11 * and limitations under the License.
12 *
13 * When distributing Covered Code, include this CDDL HEADER in each
14 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
15 * If applicable, add the following below this CDDL HEADER, with the
16 * fields enclosed by brackets "[]" replaced with your own identifying
17 * information: Portions Copyright [yyyy] [name of copyright owner]
18 *
19 * CDDL HEADER END
20 */

22 /*
23  * Copyright 2008 Sun Microsystems, Inc. All rights reserved.
24  * Use is subject to license terms.
25  */
```

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

```
28 #include <stdlib.h>
29 #include <string.h>
30 #include <errno.h>
```

```
31 #endif /* ! codereview */
32 /*
33  * calloc - allocate and clear memory block
34  */
```

```
36 void *
37 calloc(size_t num, size_t size)
38 {
39     void *mp;
40     size_t total;

42     if (num == 0 || size == 0) {
43         total = 0;
44     } else {
45         total = num * size;

47         /* check for overflow */
48         if (total / num != size) {
49             errno = ENOMEM;
50             return (NULL);
51         }
52     }
53 #endif /* ! codereview */

55     mp = malloc(total);
56     num *= size;
57     mp = malloc(num);
58     if (mp == NULL)
59         return (NULL);
```

```
58     (void) memset(mp, 0, total);
59     (void) memset(mp, 0, num);
60     return (mp);
61 }
```

```
_____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

```
330 void *
331 calloc(size_t nelem, size_t bytes)
332 {
333     void * ptr;
334     size_t size;
335
336     if (nelem == 0 || bytes == 0) {
337         size = 0;
338     } else {
339         size = nelem * bytes;
340
341         /* check for overflow */
342         if (size / nelem != bytes) {
343             errno = ENOMEM;
344             return (NULL);
345         }
346     }
347     size_t size = nelem * bytes;
348
349     ptr = malloc(size);
350     if (ptr == NULL)
351         return (NULL);
352     (void) memset(ptr, 0, size);
353
354     return (ptr);
355 }
```

unchanged portion omitted