```
**********************************************************
  181861 Wed May 27 16:12:07 2015
new/usr/src/uts/common/os/kmem.c
XXXX kmem: double-calling kmem_depot_ws_update isn't obvious
While the double-call is documented in a comment, it's not obvious what
exactly it is trying to accomplish.  The easiest way to address this is to
introduce a new function that "zeroes-out" the working set statistics to
force everything to be eligible for reaping.
**********************************************************
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */
/*
 * Copyright (c) 1994, 2010, Oracle and/or its affiliates. All rights reserved.
 * Copyright 2015 Nexenta Systems, Inc.  All rights reserved.
#endif /* ! codereview */
 */

/*
 * Kernel memory allocator, as described in the following two papers and a
 * statement about the consolidator:
 *
 * Jeff Bonwick,
 * The Slab Allocator: An Object-Caching Kernel Memory Allocator.
 * Proceedings of the Summer 1994 Usenix Conference.
 * Available as /shared/sac/PSARC/1994/028/materials/kmem.pdf.
 *
 * Jeff Bonwick and Jonathan Adams,
 * Magazines and vmem: Extending the Slab Allocator to Many CPUs and
 * Arbitrary Resources.
 * Proceedings of the 2001 Usenix Conference.
 * Available as /shared/sac/PSARC/2000/550/materials/vmem.pdf.
 *
 * kmem Slab Consolidator Big Theory Statement:
 *
 * 1. Motivation
 *
 * As stated in Bonwick94, slabs provide the following advantages over other
 * allocation structures in terms of memory fragmentation:
 *
 *  - Internal fragmentation (per-buffer wasted space) is minimal.
 *  - Severe external fragmentation (unused buffers on the free list) is
 *    unlikely.
 *
 * Segregating objects by size eliminates one source of external fragmentation,
 * and according to Bonwick:
 *
 *   The other reason that slabs reduce external fragmentation is that all
 *   objects in a slab are of the same type, so they have the same lifetime
```



```
 *   distribution. The resulting segregation of short-lived and long-lived
 *   objects at slab granularity reduces the likelihood of an entire page being
 *   held hostage due to a single long-lived allocation [Barrett93, Hanson90].
 *
 * While unlikely, severe external fragmentation remains possible. Clients that
 * allocate both short- and long-lived objects from the same cache cannot
 * anticipate the distribution of long-lived objects within the allocator's slab
 * implementation. Even a small percentage of long-lived objects distributed
 * randomly across many slabs can lead to a worst case scenario where the client
 * frees the majority of its objects and the system gets back almost none of the
 * slabs. Despite the client doing what it reasonably can to help the system
 * reclaim memory, the allocator cannot shake free enough slabs because of
 * lonely allocations stubbornly hanging on. Although the allocator is in a
 * position to diagnose the fragmentation, there is nothing that the allocator
 * by itself can do about it. It only takes a single allocated object to prevent
 * an entire slab from being reclaimed, and any object handed out by
 * kmem_cache_alloc() is by definition in the client's control. Conversely,
 * although the client is in a position to move a long-lived object, it has no
 * way of knowing if the object is causing fragmentation, and if so, where to
 * move it. A solution necessarily requires further cooperation between the
 * allocator and the client.
 *
 * 2. Move Callback
 *
 * The kmem slab consolidator therefore adds a move callback to the
 * allocator/client interface, improving worst-case external fragmentation in
 * kmem caches that supply a function to move objects from one memory location
 * to another. In a situation of low memory kmem attempts to consolidate all of
 * a cache's slabs at once; otherwise it works slowly to bring external
 * fragmentation within the 1/8 limit guaranteed for internal fragmentation,
 * thereby helping to avoid a low memory situation in the future.
 *
 * The callback has the following signature:
 *
 *   kmem_cbrc_t move(void *old, void *new, size_t size, void *user_arg)
 *
 * It supplies the kmem client with two addresses: the allocated object that
 * kmem wants to move and a buffer selected by kmem for the client to use as the
 * copy destination. The callback is kmem's way of saying "Please get off of
 * this buffer and use this one instead." kmem knows where it wants to move the
 * object in order to best reduce fragmentation. All the client needs to know
 * about the second argument (void *new) is that it is an allocated, constructed
 * object ready to take the contents of the old object. When the move function
 * is called, the system is likely to be low on memory, and the new object
 * spares the client from having to worry about allocating memory for the
 * requested move. The third argument supplies the size of the object, in case a
 * single move function handles multiple caches whose objects differ only in
 * size (such as zio_buf_512, zio_buf_1024, etc). Finally, the same optional
 * user argument passed to the constructor, destructor, and reclaim functions is
 * also passed to the move callback.
 *
 * 2.1 Setting the Move Callback
 *
 * The client sets the move callback after creating the cache and before
 * allocating from it:
 *
 *      object_cache = kmem_cache_create(...);
 *      kmem_cache_set_move(object_cache, object_move);
 *
 * 2.2 Move Callback Return Values
 *
 * Only the client knows about its own data and when is a good time to move it.
 * The client is cooperating with kmem to return unused memory to the system,
 * and kmem respectfully accepts this help at the client's convenience. When
 * asked to move an object, the client can respond with any of the following:
 *
```

```
 *   typedef enum kmem_cbrc {
 *           KMEM_CBRC_YES,
 *           KMEM_CBRC_NO,
 *           KMEM_CBRC_LATER,
 *           KMEM_CBRC_DONT_NEED,
 *           KMEM_CBRC_DONT_KNOW
 *   } kmem_cbrc_t;
 *
 * The client must not explicitly kmem_cache_free() either of the objects passed
 * to the callback, since kmem wants to free them directly to the slab layer
 * (bypassing the per-CPU magazine layer). The response tells kmem which of the
 * objects to free:
 *
 *       YES: (Did it) The client moved the object, so kmem frees the old one.
 *        NO: (Never) The client refused, so kmem frees the new object (the
 *            unused copy destination). kmem also marks the slab of the old
 *            object so as not to bother the client with further callbacks for
 *            that object as long as the slab remains on the partial slab list.
 *            (The system won't be getting the slab back as long as the
 *            immovable object holds it hostage, so there's no point in moving
 *            any of its objects.)
 *     LATER: The client is using the object and cannot move it now, so kmem
 *            frees the new object (the unused copy destination). kmem still
 *            attempts to move other objects off the slab, since it expects to
 *            succeed in clearing the slab in a later callback. The client
 *            should use LATER instead of NO if the object is likely to become
 *            movable very soon.
 * DONT_NEED: The client no longer needs the object, so kmem frees the old along
 *            with the new object (the unused copy destination). This response
 *            is the client's opportunity to be a model citizen and give back as
 *            much as it can.
 * DONT_KNOW: The client does not know about the object because
 *            a) the client has just allocated the object and not yet put it
 *               wherever it expects to find known objects
 *            b) the client has removed the object from wherever it expects to
 *               find known objects and is about to free it, or
 *            c) the client has freed the object.
 *            In all these cases (a, b, and c) kmem frees the new object (the
 *            unused copy destination) and searches for the old object in the
 *            magazine layer. If found, the object is removed from the magazine
 *            layer and freed to the slab layer so it will no longer hold the
 *            slab hostage.
 *
 * 2.3 Object States
 *
 * Neither kmem nor the client can be assumed to know the object's whereabouts
 * at the time of the callback. An object belonging to a kmem cache may be in
 * any of the following states:
 *
 * 1. Uninitialized on the slab
 * 2. Allocated from the slab but not constructed (still uninitialized)
 * 3. Allocated from the slab, constructed, but not yet ready for business
 *    (not in a valid state for the move callback)
 * 4. In use (valid and known to the client)
 * 5. About to be freed (no longer in a valid state for the move callback)
 * 6. Freed to a magazine (still constructed)
 * 7. Allocated from a magazine, not yet ready for business (not in a valid
 *    state for the move callback), and about to return to state #4
 * 8. Deconstructed on a magazine that is about to be freed
 * 9. Freed to the slab
 *
 * Since the move callback may be called at any time while the object is in any
 * of the above states (except state #1), the client needs a safe way to
 * determine whether or not it knows about the object. Specifically, the client
 * needs to know whether or not the object is in state #4, the only state in
 * which a move is valid. If the object is in any other state, the client should
```



```
 * immediately return KMEM_CBRC_DONT_KNOW, since it is unsafe to access any of
 * the object's fields.
 *
 * Note that although an object may be in state #4 when kmem initiates the move
 * request, the object may no longer be in that state by the time kmem actually
 * calls the move function. Not only does the client free objects
 * asynchronously, kmem itself puts move requests on a queue where thay are
 * pending until kmem processes them from another context. Also, objects freed
 * to a magazine appear allocated from the point of view of the slab layer, so
 * kmem may even initiate requests for objects in a state other than state #4.
 *
 * 2.3.1 Magazine Layer
 *
 * An important insight revealed by the states listed above is that the magazine
 * layer is populated only by kmem_cache_free(). Magazines of constructed
 * objects are never populated directly from the slab layer (which contains raw,
 * unconstructed objects). Whenever an allocation request cannot be satisfied
 * from the magazine layer, the magazines are bypassed and the request is
 * satisfied from the slab layer (creating a new slab if necessary). kmem calls
 * the object constructor only when allocating from the slab layer, and only in
 * response to kmem_cache_alloc() or to prepare the destination buffer passed in
 * the move callback. kmem does not preconstruct objects in anticipation of
 * kmem_cache_alloc().
 *
 * 2.3.2 Object Constructor and Destructor
 *
 * If the client supplies a destructor, it must be valid to call the destructor
 * on a newly created object (immediately after the constructor).
 *
 * 2.4 Recognizing Known Objects
 *
 * There is a simple test to determine safely whether or not the client knows
 * about a given object in the move callback. It relies on the fact that kmem
 * guarantees that the object of the move callback has only been touched by the
 * client itself or else by kmem. kmem does this by ensuring that none of the
 * cache's slabs are freed to the virtual memory (VM) subsystem while a move
 * callback is pending. When the last object on a slab is freed, if there is a
 * pending move, kmem puts the slab on a per-cache dead list and defers freeing
 * slabs on that list until all pending callbacks are completed. That way,
 * clients can be certain that the object of a move callback is in one of the
 * states listed above, making it possible to distinguish known objects (in
 * state #4) using the two low order bits of any pointer member (with the
 * exception of 'char *' or 'short *' which may not be 4-byte aligned on some
 * platforms).
 *
 * The test works as long as the client always transitions objects from state #4
 * (known, in use) to state #5 (about to be freed, invalid) by setting the low
 * order bit of the client-designated pointer member. Since kmem only writes
 * invalid memory patterns, such as 0xbaddcafe to uninitialized memory and
 * 0xdeadbeef to freed memory, any scribbling on the object done by kmem is
 * guaranteed to set at least one of the two low order bits. Therefore, given an
 * object with a back pointer to a 'container_t *o_container', the client can
 * test
 *
 *      container_t *container = object->o_container;
 *      if ((uintptr_t)container & 0x3) {
 *              return (KMEM_CBRC_DONT_KNOW);
 *      }
 *
 * Typically, an object will have a pointer to some structure with a list or
 * hash where objects from the cache are kept while in use. Assuming that the
 * client has some way of knowing that the container structure is valid and will
 * not go away during the move, and assuming that the structure includes a lock
 * to protect whatever collection is used, then the client would continue as
 * follows:
 *
```

```
256  *      // Ensure that the container structure does not go away.
257  *      if (container_hold(container) == 0) {
258  *              return (KMEM_CBRC_DONT_KNOW);
259  *      }
260  *      mutex_enter(&container->c_objects_lock);
261  *      if (container != object->o_container) {
262  *              mutex_exit(&container->c_objects_lock);
263  *              container_rele(container);
264  *              return (KMEM_CBRC_DONT_KNOW);
265  *      }
266  *
267  * At this point the client knows that the object cannot be freed as long as
268  * c_objects_lock is held. Note that after acquiring the lock, the client must
269  * recheck the o_container pointer in case the object was removed just before
270  * acquiring the lock.
271  *
272  * When the client is about to free an object, it must first remove that object
273  * from the list, hash, or other structure where it is kept. At that time, to
274  * mark the object so it can be distinguished from the remaining, known objects,
275  * the client sets the designated low order bit:
276  *
277  *      mutex_enter(&container->c_objects_lock);
278  *      object->o_container = (void *)((uintptr_t)object->o_container | 0x1);
279  *      list_remove(&container->c_objects, object);
280  *      mutex_exit(&container->c_objects_lock);
281  *
282  * In the common case, the object is freed to the magazine layer, where it may
283  * be reused on a subsequent allocation without the overhead of calling the
284  * constructor. While in the magazine it appears allocated from the point of
285  * view of the slab layer, making it a candidate for the move callback. Most
286  * objects unrecognized by the client in the move callback fall into this
287  * category and are cheaply distinguished from known objects by the test
288  * described earlier. Since recognition is cheap for the client, and searching
289  * magazines is expensive for kmem, kmem defers searching until the client first
290  * returns KMEM_CBRC_DONT_KNOW. As long as the needed effort is reasonable, kmem
291  * elsewhere does what it can to avoid bothering the client unnecessarily.
292  *
293  * Invalidating the designated pointer member before freeing the object marks
294  * the object to be avoided in the callback, and conversely, assigning a valid
295  * value to the designated pointer member after allocating the object makes the
296  * object fair game for the callback:
297  *
298  *      ... allocate object ...
299  *      ... set any initial state not set by the constructor ...
300  *
301  *      mutex_enter(&container->c_objects_lock);
302  *      list_insert_tail(&container->c_objects, object);
303  *      membar_producer();
304  *      object->o_container = container;
305  *      mutex_exit(&container->c_objects_lock);
306  *
307  * Note that everything else must be valid before setting o_container makes the
308  * object fair game for the move callback. The membar_producer() call ensures
309  * that all the object's state is written to memory before setting the pointer
310  * that transitions the object from state #3 or #7 (allocated, constructed, not
311  * yet in use) to state #4 (in use, valid). That's important because the move
312  * function has to check the validity of the pointer before it can safely
313  * acquire the lock protecting the collection where it expects to find known
314  * objects.
315  *
316  * This method of distinguishing known objects observes the usual symmetry:
317  * invalidating the designated pointer is the first thing the client does before
318  * freeing the object, and setting the designated pointer is the last thing the
319  * client does after allocating the object. Of course, the client is not
320  * required to use this method. Fundamentally, how the client recognizes known
321  * objects is completely up to the client, but this method is recommended as an
```



```
322  * efficient and safe way to take advantage of the guarantees made by kmem. If
323  * the entire object is arbitrary data without any markable bits from a suitable
324  * pointer member, then the client must find some other method, such as
325  * searching a hash table of known objects.
326  *
327  * 2.5 Preventing Objects From Moving
328  *
329  * Besides a way to distinguish known objects, the other thing that the client
330  * needs is a strategy to ensure that an object will not move while the client
331  * is actively using it. The details of satisfying this requirement tend to be
332  * highly cache-specific. It might seem that the same rules that let a client
333  * remove an object safely should also decide when an object can be moved
334  * safely. However, any object state that makes a removal attempt invalid is
335  * likely to be long-lasting for objects that the client does not expect to
336  * remove. kmem knows nothing about the object state and is equally likely (from
337  * the client's point of view) to request a move for any object in the cache,
338  * whether prepared for removal or not. Even a low percentage of objects stuck
339  * in place by unremovability will defeat the consolidator if the stuck objects
340  * are the same long-lived allocations likely to hold slabs hostage.
341  * Fundamentally, the consolidator is not aimed at common cases. Severe external
342  * fragmentation is a worst case scenario manifested as sparsely allocated
343  * slabs, by definition a low percentage of the cache's objects. When deciding
344  * what makes an object movable, keep in mind the goal of the consolidator: to
345  * bring worst-case external fragmentation within the limits guaranteed for
346  * internal fragmentation. Removability is a poor criterion if it is likely to
347  * exclude more than an insignificant percentage of objects for long periods of
348  * time.
349  *
350  * A tricky general solution exists, and it has the advantage of letting you
351  * move any object at almost any moment, practically eliminating the likelihood
352  * that an object can hold a slab hostage. However, if there is a cache-specific
353  * way to ensure that an object is not actively in use in the vast majority of
354  * cases, a simpler solution that leverages this cache-specific knowledge is
355  * preferred.
356  *
357  * 2.5.1 Cache-Specific Solution
358  *
359  * As an example of a cache-specific solution, the ZFS znode cache takes
360  * advantage of the fact that the vast majority of znodes are only being
361  * referenced from the DNLC. (A typical case might be a few hundred in active
362  * use and a hundred thousand in the DNLC.) In the move callback, after the ZFS
363  * client has established that it recognizes the znode and can access its fields
364  * safely (using the method described earlier), it then tests whether the znode
365  * is referenced by anything other than the DNLC. If so, it assumes that the
366  * znode may be in active use and is unsafe to move, so it drops its locks and
367  * returns KMEM_CBRC_LATER. The advantage of this strategy is that everywhere
368  * else znodes are used, no change is needed to protect against the possibility
369  * of the znode moving. The disadvantage is that it remains possible for an
370  * application to hold a znode slab hostage with an open file descriptor.
371  * However, this case ought to be rare and the consolidator has a way to deal
372  * with it: If the client responds KMEM_CBRC_LATER repeatedly for the same
373  * object, kmem eventually stops believing it and treats the slab as if the
374  * client had responded KMEM_CBRC_NO. Having marked the hostage slab, kmem can
375  * then focus on getting it off of the partial slab list by allocating rather
376  * than freeing all of its objects. (Either way of getting a slab off the
377  * free list reduces fragmentation.)
378  *
379  * 2.5.2 General Solution
380  *
381  * The general solution, on the other hand, requires an explicit hold everywhere
382  * the object is used to prevent it from moving. To keep the client locking
383  * strategy as uncomplicated as possible, kmem guarantees the simplifying
384  * assumption that move callbacks are sequential, even across multiple caches.
385  * Internally, a global queue processed by a single thread supports all caches
386  * implementing the callback function. No matter how many caches supply a move
387  * function, the consolidator never moves more than one object at a time, so the
```

```
388  * client does not have to worry about tricky lock ordering involving several
389  * related objects from different kmem caches.
390  *
391  * The general solution implements the explicit hold as a read-write lock, which
392  * allows multiple readers to access an object from the cache simultaneously
393  * while a single writer is excluded from moving it. A single rwlock for the
394  * entire cache would lock out all threads from using any of the cache's objects
395  * even though only a single object is being moved, so to reduce contention,
396  * the client can fan out the single rwlock into an array of rwlocks hashed by
397  * the object address, making it probable that moving one object will not
398  * prevent other threads from using a different object. The rwlock cannot be a
399  * member of the object itself, because the possibility of the object moving
400  * makes it unsafe to access any of the object's fields until the lock is
401  * acquired.
402  *
403  * Assuming a small, fixed number of locks, it's possible that multiple objects
404  * will hash to the same lock. A thread that needs to use multiple objects in
405  * the same function may acquire the same lock multiple times. Since rwlocks are
406  * reentrant for readers, and since there is never more than a single writer at
407  * a time (assuming that the client acquires the lock as a writer only when
408  * moving an object inside the callback), there would seem to be no problem.
409  * However, a client locking multiple objects in the same function must handle
410  * one case of potential deadlock: Assume that thread A needs to prevent both
411  * object 1 and object 2 from moving, and thread B, the callback, meanwhile
412  * tries to move object 3. It's possible, if objects 1, 2, and 3 all hash to the
413  * same lock, that thread A will acquire the lock for object 1 as a reader
414  * before thread B sets the lock's write-wanted bit, preventing thread A from
415  * reacquiring the lock for object 2 as a reader. Unable to make forward
416  * progress, thread A will never release the lock for object 1, resulting in
417  * deadlock.
418  *
419  * There are two ways of avoiding the deadlock just described. The first is to
420  * use rw_tryenter() rather than rw_enter() in the callback function when
421  * attempting to acquire the lock as a writer. If tryenter discovers that the
422  * same object (or another object hashed to the same lock) is already in use, it
423  * aborts the callback and returns KMEM_CBRC_LATER. The second way is to use
424  * rprwlock_t (declared in common/fs/zfs/sys/rprwlock.h) instead of rwlock_t,
425  * since it allows a thread to acquire the lock as a reader in spite of a
426  * waiting writer. This second approach insists on moving the object now, no
427  * matter how many readers the move function must wait for in order to do so,
428  * and could delay the completion of the callback indefinitely (blocking
429  * callbacks to other clients). In practice, a less insistent callback using
430  * rw_tryenter() returns KMEM_CBRC_LATER infrequently enough that there seems
431  * little reason to use anything else.
432  *
433  * Avoiding deadlock is not the only problem that an implementation using an
434  * explicit hold needs to solve. Locking the object in the first place (to
435  * prevent it from moving) remains a problem, since the object could move
436  * between the time you obtain a pointer to the object and the time you acquire
437  * the rwlock hashed to that pointer value. Therefore the client needs to
438  * recheck the value of the pointer after acquiring the lock, drop the lock if
439  * the value has changed, and try again. This requires a level of indirection:
440  * something that points to the object rather than the object itself, that the
441  * client can access safely while attempting to acquire the lock. (The object
442  * itself cannot be referenced safely because it can move at any time.)
443  * The following lock-acquisition function takes whatever is safe to reference
444  * (arg), follows its pointer to the object (using function f), and tries as
445  * often as necessary to acquire the hashed lock and verify that the object
446  * still has not moved:
447  *
448  *      object_t *
449  *      object_hold(object_f f, void *arg)
450  *      {
451  *              object_t *op;
452  *
453  *              op = f(arg);
```



```
454  *              if (op == NULL) {
455  *                      return (NULL);
456  *              }
457  *
458  *              rw_enter(OBJECT_RWLOCK(op), RW_READER);
459  *              while (op != f(arg)) {
460  *                      rw_exit(OBJECT_RWLOCK(op));
461  *                      op = f(arg);
462  *                      if (op == NULL) {
463  *                              break;
464  *                      }
465  *                      rw_enter(OBJECT_RWLOCK(op), RW_READER);
466  *              }
467  *
468  *              return (op);
469  *      }
470  *
471  * The OBJECT_RWLOCK macro hashes the object address to obtain the rwlock. The
472  * lock reacquisition loop, while necessary, almost never executes. The function
473  * pointer f (used to obtain the object pointer from arg) has the following type
474  * definition:
475  *
476  *      typedef object_t *(*object_f)(void *arg);
477  *
478  * An object_f implementation is likely to be as simple as accessing a structure
479  * member:
480  *
481  *      object_t *
482  *      s_object(void *arg)
483  *      {
484  *              something_t *sp = arg;
485  *              return (sp->s_object);
486  *      }
487  *
488  * The flexibility of a function pointer allows the path to the object to be
489  * arbitrarily complex and also supports the notion that depending on where you
490  * are using the object, you may need to get it from someplace different.
491  *
492  * The function that releases the explicit hold is simpler because it does not
493  * have to worry about the object moving:
494  *
495  *      void
496  *      object_rele(object_t *op)
497  *      {
498  *              rw_exit(OBJECT_RWLOCK(op));
499  *      }
500  *
501  * The caller is spared these details so that obtaining and releasing an
502  * explicit hold feels like a simple mutex_enter()/mutex_exit() pair. The caller
503  * of object_hold() only needs to know that the returned object pointer is valid
504  * if not NULL and that the object will not move until released.
505  *
506  * Although object_hold() prevents an object from moving, it does not prevent it
507  * from being freed. The caller must take measures before calling object_hold()
508  * (afterwards is too late) to ensure that the held object cannot be freed. The
509  * caller must do so without accessing the unsafe object reference, so any lock
510  * or reference count used to ensure the continued existence of the object must
511  * live outside the object itself.
512  *
513  * Obtaining a new object is a special case where an explicit hold is impossible
514  * for the caller. Any function that returns a newly allocated object (either as
515  * a return value, or as an in-out paramter) must return it already held; after
516  * the caller gets it is too late, since the object cannot be safely accessed
517  * without the level of indirection described earlier. The following
518  * object_alloc() example uses the same code shown earlier to transition a new
519  * object into the state of being recognized (by the client) as a known object.
```

```
 520  * The function must acquire the hold (rw_enter) before that state transition
 521  * makes the object movable:
 522  *
 523  *      static object_t *
 524  *      object_alloc(container_t *container)
 525  *      {
 526  *              object_t *object = kmem_cache_alloc(object_cache, 0);
 527  *              ... set any initial state not set by the constructor ...
 528  *              rw_enter(OBJECT_RWLOCK(object), RW_READER);
 529  *              mutex_enter(&container->c_objects_lock);
 530  *              list_insert_tail(&container->c_objects, object);
 531  *              membar_producer();
 532  *              object->o_container = container;
 533  *              mutex_exit(&container->c_objects_lock);
 534  *              return (object);
 535  *      }
 536  *
 537  * Functions that implicitly acquire an object hold (any function that calls
 538  * object_alloc() to supply an object for the caller) need to be carefully noted
 539  * so that the matching object_rele() is not neglected. Otherwise, leaked holds
 540  * prevent all objects hashed to the affected rwlocks from ever being moved.
 541  *
 542  * The pointer to a held object can be hashed to the holding rwlock even after
 543  * the object has been freed. Although it is possible to release the hold
 544  * after freeing the object, you may decide to release the hold implicitly in
 545  * whatever function frees the object, so as to release the hold as soon as
 546  * possible, and for the sake of symmetry with the function that implicitly
 547  * acquires the hold when it allocates the object. Here, object_free() releases
 548  * the hold acquired by object_alloc(). Its implicit object_rele() forms a
 549  * matching pair with object_hold():
 550  *
 551  *      void
 552  *      object_free(object_t *object)
 553  *      {
 554  *              container_t *container;
 555  *
 556  *              ASSERT(object_held(object));
 557  *              container = object->o_container;
 558  *              mutex_enter(&container->c_objects_lock);
 559  *              object->o_container =
 560  *                  (void *)((uintptr_t)object->o_container | 0x1);
 561  *              list_remove(&container->c_objects, object);
 562  *              mutex_exit(&container->c_objects_lock);
 563  *              object_rele(object);
 564  *              kmem_cache_free(object_cache, object);
 565  *      }
 566  *
 567  * Note that object_free() cannot safely accept an object pointer as an argument
 568  * unless the object is already held. Any function that calls object_free()
 569  * needs to be carefully noted since it similarly forms a matching pair with
 570  * object_hold().
 571  *
 572  * To complete the picture, the following callback function implements the
 573  * general solution by moving objects only if they are currently unheld:
 574  *
 575  *      static kmem_cbrc_t
 576  *      object_move(void *buf, void *newbuf, size_t size, void *arg)
 577  *      {
 578  *              object_t *op = buf, *np = newbuf;
 579  *              container_t *container;
 580  *
 581  *              container = op->o_container;
 582  *              if ((uintptr_t)container & 0x3) {
 583  *                      return (KMEM_CBRC_DONT_KNOW);
 584  *              }
 585  *
```



```
 586  *              // Ensure that the container structure does not go away.
 587  *              if (container_hold(container) == 0) {
 588  *                      return (KMEM_CBRC_DONT_KNOW);
 589  *              }
 590  *
 591  *              mutex_enter(&container->c_objects_lock);
 592  *              if (container != op->o_container) {
 593  *                      mutex_exit(&container->c_objects_lock);
 594  *                      container_rele(container);
 595  *                      return (KMEM_CBRC_DONT_KNOW);
 596  *              }
 597  *
 598  *              if (rw_tryenter(OBJECT_RWLOCK(op), RW_WRITER) == 0) {
 599  *                      mutex_exit(&container->c_objects_lock);
 600  *                      container_rele(container);
 601  *                      return (KMEM_CBRC_LATER);
 602  *              }
 603  *
 604  *              object_move_impl(op, np); // critical section
 605  *              rw_exit(OBJECT_RWLOCK(op));
 606  *
 607  *              op->o_container = (void *)((uintptr_t)op->o_container | 0x1);
 608  *              list_link_replace(&op->o_link_node, &np->o_link_node);
 609  *              mutex_exit(&container->c_objects_lock);
 610  *              container_rele(container);
 611  *              return (KMEM_CBRC_YES);
 612  *      }
 613  *
 614  * Note that object_move() must invalidate the designated o_container pointer of
 615  * the old object in the same way that object_free() does, since kmem will free
 616  * the object in response to the KMEM_CBRC_YES return value.
 617  *
 618  * The lock order in object_move() differs from object_alloc(), which locks
 619  * OBJECT_RWLOCK first and &container->c_objects_lock second, but as long as the
 620  * callback uses rw_tryenter() (preventing the deadlock described earlier), it's
 621  * not a problem. Holding the lock on the object list in the example above
 622  * through the entire callback not only prevents the object from going away, it
 623  * also allows you to lock the list elsewhere and know that none of its elements
 624  * will move during iteration.
 625  *
 626  * Adding an explicit hold everywhere an object from the cache is used is tricky
 627  * and involves much more change to client code than a cache-specific solution
 628  * that leverages existing state to decide whether or not an object is
 629  * movable. However, this approach has the advantage that no object remains
 630  * immovable for any significant length of time, making it extremely unlikely
 631  * that long-lived allocations can continue holding slabs hostage; and it works
 632  * for any cache.
 633  *
 634  * 3. Consolidator Implementation
 635  *
 636  * Once the client supplies a move function that a) recognizes known objects and
 637  * b) avoids moving objects that are actively in use, the remaining work is up
 638  * to the consolidator to decide which objects to move and when to issue
 639  * callbacks.
 640  *
 641  * The consolidator relies on the fact that a cache's slabs are ordered by
 642  * usage. Each slab has a fixed number of objects. Depending on the slab's
 643  * "color" (the offset of the first object from the beginning of the slab;
 644  * offsets are staggered to mitigate false sharing of cache lines) it is either
 645  * the maximum number of objects per slab determined at cache creation time or
 646  * else the number closest to the maximum that fits within the space remaining
 647  * after the initial offset. A completely allocated slab may contribute some
 648  * internal fragmentation (per-slab overhead) but no external fragmentation, so
 649  * it is of no interest to the consolidator. At the other extreme, slabs whose
 650  * objects have all been freed to the slab are released to the virtual memory
 651  * (VM) subsystem (objects freed to magazines are still allocated as far as the
```

```
652  * slab is concerned). External fragmentation exists when there are slabs
653  * somewhere between these extremes. A partial slab has at least one but not all
654  * of its objects allocated. The more partial slabs, and the fewer allocated
655  * objects on each of them, the higher the fragmentation. Hence the
656  * consolidator's overall strategy is to reduce the number of partial slabs by
657  * moving allocated objects from the least allocated slabs to the most allocated
658  * slabs.
659  *
660  * Partial slabs are kept in an AVL tree ordered by usage. Completely allocated
661  * slabs are kept separately in an unordered list. Since the majority of slabs
662  * tend to be completely allocated (a typical unfragmented cache may have
663  * thousands of complete slabs and only a single partial slab), separating
664  * complete slabs improves the efficiency of partial slab ordering, since the
665  * complete slabs do not affect the depth or balance of the AVL tree. This
666  * ordered sequence of partial slabs acts as a "free list" supplying objects for
667  * allocation requests.
668  *
669  * Objects are always allocated from the first partial slab in the free list,
670  * where the allocation is most likely to eliminate a partial slab (by
671  * completely allocating it). Conversely, when a single object from a completely
672  * allocated slab is freed to the slab, that slab is added to the front of the
673  * free list. Since most free list activity involves highly allocated slabs
674  * coming and going at the front of the list, slabs tend naturally toward the
675  * ideal order: highly allocated at the front, sparsely allocated at the back.
676  * Slabs with few allocated objects are likely to become completely free if they
677  * keep a safe distance away from the front of the free list. Slab misorders
678  * interfere with the natural tendency of slabs to become completely free or
679  * completely allocated. For example, a slab with a single allocated object
680  * needs only a single free to escape the cache; its natural desire is
681  * frustrated when it finds itself at the front of the list where a second
682  * allocation happens just before the free could have released it. Another slab
683  * with all but one object allocated might have supplied the buffer instead, so
684  * that both (as opposed to neither) of the slabs would have been taken off the
685  * free list.
686  *
687  * Although slabs tend naturally toward the ideal order, misorders allowed by a
688  * simple list implementation defeat the consolidator's strategy of merging
689  * least- and most-allocated slabs. Without an AVL tree to guarantee order, kmem
690  * needs another way to fix misorders to optimize its callback strategy. One
691  * approach is to periodically scan a limited number of slabs, advancing a
692  * marker to hold the current scan position, and to move extreme misorders to
693  * the front or back of the free list and to the front or back of the current
694  * scan range. By making consecutive scan ranges overlap by one slab, the least
695  * allocated slab in the current range can be carried along from the end of one
696  * scan to the start of the next.
697  *
698  * Maintaining partial slabs in an AVL tree relieves kmem of this additional
699  * task, however. Since most of the cache's activity is in the magazine layer,
700  * and allocations from the slab layer represent only a startup cost, the
701  * overhead of maintaining a balanced tree is not a significant concern compared
702  * to the opportunity of reducing complexity by eliminating the partial slab
703  * scanner just described. The overhead of an AVL tree is minimized by
704  * maintaining only partial slabs in the tree and keeping completely allocated
705  * slabs separately in a list. To avoid increasing the size of the slab
706  * structure the AVL linkage pointers are reused for the slab's list linkage,
707  * since the slab will always be either partial or complete, never stored both
708  * ways at the same time. To further minimize the overhead of the AVL tree the
709  * compare function that orders partial slabs by usage divides the range of
710  * allocated object counts into bins such that counts within the same bin are
711  * considered equal. Binning partial slabs makes it less likely that allocating
712  * or freeing a single object will change the slab's order, requiring a tree
713  * reinsertion (an avl_remove() followed by an avl_add(), both potentially
714  * requiring some rebalancing of the tree). Allocation counts closest to
715  * completely free and completely allocated are left unbinned (finely sorted) to
716  * better support the consolidator's strategy of merging slabs at either
717  * extreme.
```



```
718  *
719  * 3.1 Assessing Fragmentation and Selecting Candidate Slabs
720  *
721  * The consolidator piggybacks on the kmem maintenance thread and is called on
722  * the same interval as kmem_cache_update(), once per cache every fifteen
723  * seconds. kmem maintains a running count of unallocated objects in the slab
724  * layer (cache_bufslab). The consolidator checks whether that number exceeds
725  * 12.5% (1/8) of the total objects in the cache (cache_buftotal), and whether
726  * there is a significant number of slabs in the cache (arbitrarily a minimum
727  * 101 total slabs). Unused objects that have fallen out of the magazine layer's
728  * working set are included in the assessment, and magazines in the depot are
729  * reaped if those objects would lift cache_bufslab above the fragmentation
730  * threshold. Once the consolidator decides that a cache is fragmented, it looks
731  * for a candidate slab to reclaim, starting at the end of the partial slab free
732  * list and scanning backwards. At first the consolidator is choosy: only a slab
733  * with fewer than 12.5% (1/8) of its objects allocated qualifies (or else a
734  * single allocated object, regardless of percentage). If there is difficulty
735  * finding a candidate slab, kmem raises the allocation threshold incrementally,
736  * up to a maximum 87.5% (7/8), so that eventually the consolidator will reduce
737  * external fragmentation (unused objects on the free list) below 12.5% (1/8),
738  * even in the worst case of every slab in the cache being almost 7/8 allocated.
739  * The threshold can also be lowered incrementally when candidate slabs are easy
740  * to find, and the threshold is reset to the minimum 1/8 as soon as the cache
741  * is no longer fragmented.
742  *
743  * 3.2 Generating Callbacks
744  *
745  * Once an eligible slab is chosen, a callback is generated for every allocated
746  * object on the slab, in the hope that the client will move everything off the
747  * slab and make it reclaimable. Objects selected as move destinations are
748  * chosen from slabs at the front of the free list. Assuming slabs in the ideal
749  * order (most allocated at the front, least allocated at the back) and a
750  * cooperative client, the consolidator will succeed in removing slabs from both
751  * ends of the free list, completely allocating on the one hand and completely
752  * freeing on the other. Objects selected as move destinations are allocated in
753  * the kmem maintenance thread where move requests are enqueued. A separate
754  * callback thread removes pending callbacks from the queue and calls the
755  * client. The separate thread ensures that client code (the move function) does
756  * not interfere with internal kmem maintenance tasks. A map of pending
757  * callbacks keyed by object address (the object to be moved) is checked to
758  * ensure that duplicate callbacks are not generated for the same object.
759  * Allocating the move destination (the object to move to) prevents subsequent
760  * callbacks from selecting the same destination as an earlier pending callback.
761  *
762  * Move requests can also be generated by kmem_cache_reap() when the system is
763  * desperate for memory and by kmem_cache_move_notify(), called by the client to
764  * notify kmem that a move refused earlier with KMEM_CBRC_LATER is now possible.
765  * The map of pending callbacks is protected by the same lock that protects the
766  * slab layer.
767  *
768  * When the system is desperate for memory, kmem does not bother to determine
769  * whether or not the cache exceeds the fragmentation threshold, but tries to
770  * consolidate as many slabs as possible. Normally, the consolidator chews
771  * slowly, one sparsely allocated slab at a time during each maintenance
772  * interval that the cache is fragmented. When desperate, the consolidator
773  * starts at the last partial slab and enqueues callbacks for every allocated
774  * object on every partial slab, working backwards until it reaches the first
775  * partial slab. The first partial slab, meanwhile, advances in pace with the
776  * consolidator as allocations to supply move destinations for the enqueued
777  * callbacks use up the highly allocated slabs at the front of the free list.
778  * Ideally, the overgrown free list collapses like an accordion, starting at
779  * both ends and ending at the center with a single partial slab.
780  *
781  * 3.3 Client Responses
782  *
783  * When the client returns KMEM_CBRC_NO in response to the move callback, kmem
```

```
 * marks the slab that supplied the stuck object non-reclaimable and moves it to
 * front of the free list. The slab remains marked as long as it remains on the
 * free list, and it appears more allocated to the partial slab compare function
 * than any unmarked slab, no matter how many of its objects are allocated.
 * Since even one immovable object ties up the entire slab, the goal is to
 * completely allocate any slab that cannot be completely freed. kmem does not
 * bother generating callbacks to move objects from a marked slab unless the
 * system is desperate.
 *
 * When the client responds KMEM_CBRC_LATER, kmem increments a count for the
 * slab. If the client responds LATER too many times, kmem disbelieves and
 * treats the response as a NO. The count is cleared when the slab is taken off
 * the partial slab list or when the client moves one of the slab's objects.
 *
 * 4. Observability
 *
 * A kmem cache's external fragmentation is best observed with 'mdb -k' using
 * the ::kmem_slabs dcmd. For a complete description of the command, enter
 * '::help kmem_slabs' at the mdb prompt.
 */

#include <sys/kmem_impl.h>
#include <sys/vmem_impl.h>
#include <sys/param.h>
#include <sys/sysmacros.h>
#include <sys/vm.h>
#include <sys/proc.h>
#include <sys/tuneable.h>
#include <sys/systm.h>
#include <sys/cmn_err.h>
#include <sys/debug.h>
#include <sys/sdt.h>
#include <sys/mutex.h>
#include <sys/bitmap.h>
#include <sys/atomic.h>
#include <sys/kobj.h>
#include <sys/disp.h>
#include <vm/seg_kmem.h>
#include <sys/log.h>
#include <sys/callb.h>
#include <sys/taskq.h>
#include <sys/modctl.h>
#include <sys/reboot.h>
#include <sys/id32.h>
#include <sys/zone.h>
#include <sys/netstack.h>
#ifdef	DEBUG
#include <sys/random.h>
#endif

extern void streams_msg_init(void);
extern int segkp_fromheap;
extern void segkp_cache_free(void);
extern int callout_init_done;

struct kmem_cache_kstat {
        kstat_named_t   kmc_buf_size;
        kstat_named_t   kmc_align;
        kstat_named_t   kmc_chunk_size;
        kstat_named_t   kmc_slab_size;
        kstat_named_t   kmc_alloc;
        kstat_named_t   kmc_alloc_fail;
        kstat_named_t   kmc_free;
        kstat_named_t   kmc_depot_alloc;
        kstat_named_t   kmc_depot_free;
        kstat_named_t   kmc_depot_contention;
```



```
        kstat_named_t   kmc_slab_alloc;
        kstat_named_t   kmc_slab_free;
        kstat_named_t   kmc_buf_constructed;
        kstat_named_t   kmc_buf_avail;
        kstat_named_t   kmc_buf_inuse;
        kstat_named_t   kmc_buf_total;
        kstat_named_t   kmc_buf_max;
        kstat_named_t   kmc_slab_create;
        kstat_named_t   kmc_slab_destroy;
        kstat_named_t   kmc_vmem_source;
        kstat_named_t   kmc_hash_size;
        kstat_named_t   kmc_hash_lookup_depth;
        kstat_named_t   kmc_hash_rescale;
        kstat_named_t   kmc_full_magazines;
        kstat_named_t   kmc_empty_magazines;
        kstat_named_t   kmc_magazine_size;
        kstat_named_t   kmc_reap; /* number of kmem_cache_reap() calls */
        kstat_named_t   kmc_defrag; /* attempts to defrag all partial slabs */
        kstat_named_t   kmc_scan; /* attempts to defrag one partial slab */
        kstat_named_t   kmc_move_callbacks; /* sum of yes, no, later, dn, dk */
        kstat_named_t   kmc_move_yes;
        kstat_named_t   kmc_move_no;
        kstat_named_t   kmc_move_later;
        kstat_named_t   kmc_move_dont_need;
        kstat_named_t   kmc_move_dont_know; /* obj unrecognized by client ... */
        kstat_named_t   kmc_move_hunt_found; /* ... but found in mag layer */
        kstat_named_t   kmc_move_slabs_freed; /* slabs freed by consolidator */
        kstat_named_t   kmc_move_reclaimable; /* buffers, if consolidator ran */
} kmem_cache_kstat = {
        { "buf_size",           KSTAT_DATA_UINT64 },
        { "align",              KSTAT_DATA_UINT64 },
        { "chunk_size",         KSTAT_DATA_UINT64 },
        { "slab_size",          KSTAT_DATA_UINT64 },
        { "alloc",              KSTAT_DATA_UINT64 },
        { "alloc_fail",         KSTAT_DATA_UINT64 },
        { "free",               KSTAT_DATA_UINT64 },
        { "depot_alloc",        KSTAT_DATA_UINT64 },
        { "depot_free",         KSTAT_DATA_UINT64 },
        { "depot_contention",   KSTAT_DATA_UINT64 },
        { "slab_alloc",         KSTAT_DATA_UINT64 },
        { "slab_free",          KSTAT_DATA_UINT64 },
        { "buf_constructed",    KSTAT_DATA_UINT64 },
        { "buf_avail",          KSTAT_DATA_UINT64 },
        { "buf_inuse",          KSTAT_DATA_UINT64 },
        { "buf_total",          KSTAT_DATA_UINT64 },
        { "buf_max",            KSTAT_DATA_UINT64 },
        { "slab_create",        KSTAT_DATA_UINT64 },
        { "slab_destroy",       KSTAT_DATA_UINT64 },
        { "vmem_source",        KSTAT_DATA_UINT64 },
        { "hash_size",          KSTAT_DATA_UINT64 },
        { "hash_lookup_depth",  KSTAT_DATA_UINT64 },
        { "hash_rescale",       KSTAT_DATA_UINT64 },
        { "full_magazines",     KSTAT_DATA_UINT64 },
        { "empty_magazines",    KSTAT_DATA_UINT64 },
        { "magazine_size",      KSTAT_DATA_UINT64 },
        { "reap",               KSTAT_DATA_UINT64 },
        { "defrag",             KSTAT_DATA_UINT64 },
        { "scan",               KSTAT_DATA_UINT64 },
        { "move_callbacks",     KSTAT_DATA_UINT64 },
        { "move_yes",           KSTAT_DATA_UINT64 },
        { "move_no",            KSTAT_DATA_UINT64 },
        { "move_later",         KSTAT_DATA_UINT64 },
        { "move_dont_need",     KSTAT_DATA_UINT64 },
        { "move_dont_know",     KSTAT_DATA_UINT64 },
        { "move_hunt_found",    KSTAT_DATA_UINT64 },
        { "move_slabs_freed",   KSTAT_DATA_UINT64 },
```

```
916         { "move_reclaimable",   KSTAT_DATA_UINT64 },
917 };

919 static kmutex_t kmem_cache_kstat_lock;

921 /*
922  * The default set of caches to back kmem_alloc().
923  * These sizes should be reevaluated periodically.
924  *
925  * We want allocations that are multiples of the coherency granularity
926  * (64 bytes) to be satisfied from a cache which is a multiple of 64
927  * bytes, so that it will be 64-byte aligned.  For all multiples of 64,
928  * the next kmem_cache_size greater than or equal to it must be a
929  * multiple of 64.
930  *
931  * We split the table into two sections:  size <= 4k and size > 4k.  This
932  * saves a lot of space and cache footprint in our cache tables.
933  */
934 static const int kmem_alloc_sizes[] = {
935         1 * 8,
936         2 * 8,
937         3 * 8,
938         4 * 8,          5 * 8,          6 * 8,          7 * 8,
939         4 * 16,         5 * 16,         6 * 16,         7 * 16,
940         4 * 32,         5 * 32,         6 * 32,         7 * 32,
941         4 * 64,         5 * 64,         6 * 64,         7 * 64,
942         4 * 128,        5 * 128,        6 * 128,        7 * 128,
943         P2ALIGN(8192 / 7, 64),
944         P2ALIGN(8192 / 6, 64),
945         P2ALIGN(8192 / 5, 64),
946         P2ALIGN(8192 / 4, 64),
947         P2ALIGN(8192 / 3, 64),
948         P2ALIGN(8192 / 2, 64),
949 };

951 static const int kmem_big_alloc_sizes[] = {
952         2 * 4096,       3 * 4096,
953         2 * 8192,       3 * 8192,
954         4 * 8192,       5 * 8192,       6 * 8192,       7 * 8192,
955         8 * 8192,       9 * 8192,       10 * 8192,      11 * 8192,
956         12 * 8192,      13 * 8192,      14 * 8192,      15 * 8192,
957         16 * 8192
958 };

960 #define KMEM_MAXBUF             4096
961 #define KMEM_BIG_MAXBUF_32BIT   32768
962 #define KMEM_BIG_MAXBUF         131072

964 #define KMEM_BIG_MULTIPLE       4096    /* big_alloc_sizes must be a multiple */
965 #define KMEM_BIG_SHIFT          12      /* lg(KMEM_BIG_MULTIPLE) */

967 static kmem_cache_t *kmem_alloc_table[KMEM_MAXBUF >> KMEM_ALIGN_SHIFT];
968 static kmem_cache_t *kmem_big_alloc_table[KMEM_BIG_MAXBUF >> KMEM_BIG_SHIFT];

970 #define KMEM_ALLOC_TABLE_MAX    (KMEM_MAXBUF >> KMEM_ALIGN_SHIFT)
971 static size_t kmem_big_alloc_table_max = 0;     /* # of filled elements */

973 static kmem_magtype_t kmem_magtype[] = {
974         { 1,    8,      3200,   65536   },
975         { 3,    16,     256,    32768   },
976         { 7,    32,     64,     16384   },
977         { 15,   64,     0,      8192    },
978         { 31,   64,     0,      4096    },
979         { 47,   64,     0,      2048    },
980         { 63,   64,     0,      1024    },
981         { 95,   64,     0,      512     },
```



```
982         { 143,  64,     0,      0       },
983 };

985 static uint32_t kmem_reaping;
986 static uint32_t kmem_reaping_idspace;

988 /*
989  * kmem tunables
990  */
991 clock_t kmem_reap_interval;     /* cache reaping rate [15 * HZ ticks] */
992 int kmem_depot_contention = 3;  /* max failed tryenters per real interval */
993 pgcnt_t kmem_reapahead = 0;     /* start reaping N pages before pageout */
994 int kmem_panic = 1;             /* whether to panic on error */
995 int kmem_logging = 1;           /* kmem_log_enter() override */
996 uint32_t kmem_mtbf = 0;         /* mean time between failures [default: off] */
997 size_t kmem_transaction_log_size; /* transaction log size [2% of memory] */
998 size_t kmem_content_log_size;   /* content log size [2% of memory] */
999 size_t kmem_failure_log_size;   /* failure log [4 pages per CPU] */
1000 size_t kmem_slab_log_size;      /* slab create log [4 pages per CPU] */
1001 size_t kmem_content_maxsave = 256; /* KMF_CONTENTS max bytes to log */
1002 size_t kmem_lite_minsize = 0;   /* minimum buffer size for KMF_LITE */
1003 size_t kmem_lite_maxalign = 1024; /* maximum buffer alignment for KMF_LITE */
1004 int kmem_lite_pcs = 4;          /* number of PCs to store in KMF_LITE mode */
1005 size_t kmem_maxverify;          /* maximum bytes to inspect in debug routines */
1006 size_t kmem_minfirewall;        /* hardware-enforced redzone threshold */

1008 #ifdef _LP64
1009 size_t  kmem_max_cached = KMEM_BIG_MAXBUF;      /* maximum kmem_alloc cache */
1010 #else
1011 size_t  kmem_max_cached = KMEM_BIG_MAXBUF_32BIT; /* maximum kmem_alloc cache */
1012 #endif

1014 #ifdef DEBUG
1015 int kmem_flags = KMF_AUDIT | KMF_DEADBEEF | KMF_REDZONE | KMF_CONTENTS;
1016 #else
1017 int kmem_flags = 0;
1018 #endif
1019 int kmem_ready;

1021 static kmem_cache_t     *kmem_slab_cache;
1022 static kmem_cache_t     *kmem_bufctl_cache;
1023 static kmem_cache_t     *kmem_bufctl_audit_cache;

1025 static kmutex_t         kmem_cache_lock;        /* inter-cache linkage only */
1026 static list_t           kmem_caches;

1028 static taskq_t          *kmem_taskq;
1029 static kmutex_t         kmem_flags_lock;
1030 static vmem_t           *kmem_metadata_arena;
1031 static vmem_t           *kmem_msb_arena;        /* arena for metadata caches */
1032 static vmem_t           *kmem_cache_arena;
1033 static vmem_t           *kmem_hash_arena;
1034 static vmem_t           *kmem_log_arena;
1035 static vmem_t           *kmem_oversize_arena;
1036 static vmem_t           *kmem_va_arena;
1037 static vmem_t           *kmem_default_arena;
1038 static vmem_t           *kmem_firewall_va_arena;
1039 static vmem_t           *kmem_firewall_arena;

1041 /*
1042  * Define KMEM_STATS to turn on statistic gathering. By default, it is only
1043  * turned on when DEBUG is also defined.
1044  */
1045 #ifdef  DEBUG
1046 #define KMEM_STATS
1047 #endif  /* DEBUG */
```

```
1049 #ifdef  KMEM_STATS
1050 #define KMEM_STAT_ADD(stat)                     ((stat)++)
1051 #define KMEM_STAT_COND_ADD(cond, stat)          ((void) (!(cond) || (stat)++))
1052 #else
1053 #define KMEM_STAT_ADD(stat)                     /* nothing */
1054 #define KMEM_STAT_COND_ADD(cond, stat)          /* nothing */
1055 #endif  /* KMEM_STATS */

1057 /*
1058  * kmem slab consolidator thresholds (tunables)
1059  */
1060 size_t kmem_frag_minslabs = 101;        /* minimum total slabs */
1061 size_t kmem_frag_numer = 1;             /* free buffers (numerator) */
1062 size_t kmem_frag_denom = KMEM_VOID_FRACTION; /* buffers (denominator) */
1063 /*
1064  * Maximum number of slabs from which to move buffers during a single
1065  * maintenance interval while the system is not low on memory.
1066  */
1067 size_t kmem_reclaim_max_slabs = 1;
1068 /*
1069  * Number of slabs to scan backwards from the end of the partial slab list
1070  * when searching for buffers to relocate.
1071  */
1072 size_t kmem_reclaim_scan_range = 12;

1074 #ifdef  KMEM_STATS
1075 static struct {
1076         uint64_t kms_callbacks;
1077         uint64_t kms_yes;
1078         uint64_t kms_no;
1079         uint64_t kms_later;
1080         uint64_t kms_dont_need;
1081         uint64_t kms_dont_know;
1082         uint64_t kms_hunt_found_mag;
1083         uint64_t kms_hunt_found_slab;
1084         uint64_t kms_hunt_alloc_fail;
1085         uint64_t kms_hunt_lucky;
1086         uint64_t kms_notify;
1087         uint64_t kms_notify_callbacks;
1088         uint64_t kms_disbelief;
1089         uint64_t kms_already_pending;
1090         uint64_t kms_callback_alloc_fail;
1091         uint64_t kms_callback_taskq_fail;
1092         uint64_t kms_endscan_slab_dead;
1093         uint64_t kms_endscan_slab_destroyed;
1094         uint64_t kms_endscan_nomem;
1095         uint64_t kms_endscan_refcnt_changed;
1096         uint64_t kms_endscan_nomove_changed;
1097         uint64_t kms_endscan_freelist;
1098         uint64_t kms_avl_update;
1099         uint64_t kms_avl_noupdate;
1100         uint64_t kms_no_longer_reclaimable;
1101         uint64_t kms_notify_no_longer_reclaimable;
1102         uint64_t kms_notify_slab_dead;
1103         uint64_t kms_notify_slab_destroyed;
1104         uint64_t kms_alloc_fail;
1105         uint64_t kms_constructor_fail;
1106         uint64_t kms_dead_slabs_freed;
1107         uint64_t kms_defrags;
1108         uint64_t kms_scans;
1109         uint64_t kms_scan_depot_ws_reaps;
1110         uint64_t kms_debug_reaps;
1111         uint64_t kms_debug_scans;
1112 } kmem_move_stats;
1113 #endif  /* KMEM_STATS */
```



```
1115 /* consolidator knobs */
1116 static boolean_t kmem_move_noreap;
1117 static boolean_t kmem_move_blocked;
1118 static boolean_t kmem_move_fulltilt;
1119 static boolean_t kmem_move_any_partial;

1121 #ifdef  DEBUG
1122 /*
1123  * kmem consolidator debug tunables:
1124  * Ensure code coverage by occasionally running the consolidator even when the
1125  * caches are not fragmented (they may never be). These intervals are mean time
1126  * in cache maintenance intervals (kmem_cache_update).
1127  */
1128 uint32_t kmem_mtb_move = 60;    /* defrag 1 slab (~15min) */
1129 uint32_t kmem_mtb_reap = 1800;  /* defrag all slabs (~7.5hrs) */
1130 #endif  /* DEBUG */

1132 static kmem_cache_t     *kmem_defrag_cache;
1133 static kmem_cache_t     *kmem_move_cache;
1134 static taskq_t          *kmem_move_taskq;

1136 static void kmem_cache_scan(kmem_cache_t *);
1137 static void kmem_cache_defrag(kmem_cache_t *);
1138 static void kmem_slab_prefill(kmem_cache_t *, kmem_slab_t *);


1141 kmem_log_header_t       *kmem_transaction_log;
1142 kmem_log_header_t       *kmem_content_log;
1143 kmem_log_header_t       *kmem_failure_log;
1144 kmem_log_header_t       *kmem_slab_log;

1146 static int              kmem_lite_count; /* # of PCs in kmem_buftag_lite_t */

1148 #define KMEM_BUFTAG_LITE_ENTER(bt, count, caller)                       \
1149         if ((count) > 0) {                                              \
1150                 pc_t *_s = ((kmem_buftag_lite_t *)(bt))->bt_history;    \
1151                 pc_t *_e;                                               \
1152                 /* memmove() the old entries down one notch */          \
1153                 for (_e = &_s[(count) - 1]; _e > _s; _e--)              \
1154                         *_e = *(_e - 1);                                \
1155                 *_s = (uintptr_t)(caller);                              \
1156         }

1158 #define KMERR_MODIFIED  0       /* buffer modified while on freelist */
1159 #define KMERR_REDZONE   1       /* redzone violation (write past end of buf) */
1160 #define KMERR_DUPFREE   2       /* freed a buffer twice */
1161 #define KMERR_BADADDR   3       /* freed a bad (unallocated) address */
1162 #define KMERR_BADBUFTAG 4       /* buftag corrupted */
1163 #define KMERR_BADBUFCTL 5       /* bufctl corrupted */
1164 #define KMERR_BADCACHE  6       /* freed a buffer to the wrong cache */
1165 #define KMERR_BADSIZE   7       /* alloc size != free size */
1166 #define KMERR_BADBASE   8       /* buffer base address wrong */

1168 struct {
1169         hrtime_t        kmp_timestamp;  /* timestamp of panic */
1170         int             kmp_error;      /* type of kmem error */
1171         void            *kmp_buffer;    /* buffer that induced panic */
1172         void            *kmp_realbuf;   /* real start address for buffer */
1173         kmem_cache_t    *kmp_cache;     /* buffer's cache according to client */
1174         kmem_cache_t    *kmp_realcache; /* actual cache containing buffer */
1175         kmem_slab_t     *kmp_slab;      /* slab accoring to kmem_findslab() */
1176         kmem_bufctl_t   *kmp_bufctl;    /* bufctl */
1177 } kmem_panic_info;
```

```
1180 static void
1181 copy_pattern(uint64_t pattern, void *buf_arg, size_t size)
1182 {
1183         uint64_t *bufend = (uint64_t *)((char *)buf_arg + size);
1184         uint64_t *buf = buf_arg;
1185
1186         while (buf < bufend)
1187                 *buf++ = pattern;
1188 }
1189
1190 static void *
1191 verify_pattern(uint64_t pattern, void *buf_arg, size_t size)
1192 {
1193         uint64_t *bufend = (uint64_t *)((char *)buf_arg + size);
1194         uint64_t *buf;
1195
1196         for (buf = buf_arg; buf < bufend; buf++)
1197                 if (*buf != pattern)
1198                         return (buf);
1199         return (NULL);
1200 }
1201
1202 static void *
1203 verify_and_copy_pattern(uint64_t old, uint64_t new, void *buf_arg, size_t size)
1204 {
1205         uint64_t *bufend = (uint64_t *)((char *)buf_arg + size);
1206         uint64_t *buf;
1207
1208         for (buf = buf_arg; buf < bufend; buf++) {
1209                 if (*buf != old) {
1210                         copy_pattern(old, buf_arg,
1211                             (char *)buf - (char *)buf_arg);
1212                         return (buf);
1213                 }
1214                 *buf = new;
1215         }
1216
1217         return (NULL);
1218 }
1219
1220 static void
1221 kmem_cache_applyall(void (*func)(kmem_cache_t *), taskq_t *tq, int tqflag)
1222 {
1223         kmem_cache_t *cp;
1224
1225         mutex_enter(&kmem_cache_lock);
1226         for (cp = list_head(&kmem_caches); cp != NULL;
1227             cp = list_next(&kmem_caches, cp))
1228                 if (tq != NULL)
1229                         (void) taskq_dispatch(tq, (task_func_t *)func, cp,
1230                             tqflag);
1231                 else
1232                         func(cp);
1233         mutex_exit(&kmem_cache_lock);
1234 }
1235
1236 static void
1237 kmem_cache_applyall_id(void (*func)(kmem_cache_t *), taskq_t *tq, int tqflag)
1238 {
1239         kmem_cache_t *cp;
1240
1241         mutex_enter(&kmem_cache_lock);
1242         for (cp = list_head(&kmem_caches); cp != NULL;
1243             cp = list_next(&kmem_caches, cp)) {
1244                 if (!(cp->cache_cflags & KMC_IDENTIFIER))
1245                         continue;
```



```
1246                 if (tq != NULL)
1247                         (void) taskq_dispatch(tq, (task_func_t *)func, cp,
1248                             tqflag);
1249                 else
1250                         func(cp);
1251         }
1252         mutex_exit(&kmem_cache_lock);
1253 }
1254
1255 /*
1256  * Debugging support.  Given a buffer address, find its slab.
1257  */
1258 static kmem_slab_t *
1259 kmem_findslab(kmem_cache_t *cp, void *buf)
1260 {
1261         kmem_slab_t *sp;
1262
1263         mutex_enter(&cp->cache_lock);
1264         for (sp = list_head(&cp->cache_complete_slabs); sp != NULL;
1265             sp = list_next(&cp->cache_complete_slabs, sp)) {
1266                 if (KMEM_SLAB_MEMBER(sp, buf)) {
1267                         mutex_exit(&cp->cache_lock);
1268                         return (sp);
1269                 }
1270         }
1271         for (sp = avl_first(&cp->cache_partial_slabs); sp != NULL;
1272             sp = AVL_NEXT(&cp->cache_partial_slabs, sp)) {
1273                 if (KMEM_SLAB_MEMBER(sp, buf)) {
1274                         mutex_exit(&cp->cache_lock);
1275                         return (sp);
1276                 }
1277         }
1278         mutex_exit(&cp->cache_lock);
1279
1280         return (NULL);
1281 }
1282
1283 static void
1284 kmem_error(int error, kmem_cache_t *cparg, void *bufarg)
1285 {
1286         kmem_buftag_t *btp = NULL;
1287         kmem_bufctl_t *bcp = NULL;
1288         kmem_cache_t *cp = cparg;
1289         kmem_slab_t *sp;
1290         uint64_t *off;
1291         void *buf = bufarg;
1292
1293         kmem_logging = 0;       /* stop logging when a bad thing happens */
1294
1295         kmem_panic_info.kmp_timestamp = gethrtime();
1296
1297         sp = kmem_findslab(cp, buf);
1298         if (sp == NULL) {
1299                 for (cp = list_tail(&kmem_caches); cp != NULL;
1300                     cp = list_prev(&kmem_caches, cp)) {
1301                         if ((sp = kmem_findslab(cp, buf)) != NULL)
1302                                 break;
1303                 }
1304         }
1305
1306         if (sp == NULL) {
1307                 cp = NULL;
1308                 error = KMERR_BADADDR;
1309         } else {
1310                 if (cp != cparg)
1311                         error = KMERR_BADCACHE;
```

```
1312                else
1313                        buf = (char *)bufarg - ((uintptr_t)bufarg -
1314                            (uintptr_t)sp->slab_base) % cp->cache_chunksize;
1315                if (buf != bufarg)
1316                        error = KMERR_BADBASE;
1317                if (cp->cache_flags & KMF_BUFTAG)
1318                        btp = KMEM_BUFTAG(cp, buf);
1319                if (cp->cache_flags & KMF_HASH) {
1320                        mutex_enter(&cp->cache_lock);
1321                        for (bcp = *KMEM_HASH(cp, buf); bcp; bcp = bcp->bc_next)
1322                                if (bcp->bc_addr == buf)
1323                                        break;
1324                        mutex_exit(&cp->cache_lock);
1325                        if (bcp == NULL && btp != NULL)
1326                                bcp = btp->bt_bufctl;
1327                        if (kmem_findslab(cp->cache_bufctl_cache, bcp) ==
1328                            NULL || P2PHASE((uintptr_t)bcp, KMEM_ALIGN) ||
1329                            bcp->bc_addr != buf) {
1330                                error = KMERR_BADBUFCTL;
1331                                bcp = NULL;
1332                        }
1333                }
1334        }

1336        kmem_panic_info.kmp_error = error;
1337        kmem_panic_info.kmp_buffer = bufarg;
1338        kmem_panic_info.kmp_realbuf = buf;
1339        kmem_panic_info.kmp_cache = cparg;
1340        kmem_panic_info.kmp_realcache = cp;
1341        kmem_panic_info.kmp_slab = sp;
1342        kmem_panic_info.kmp_bufctl = bcp;

1344        printf("kernel memory allocator: ");

1346        switch (error) {

1348        case KMERR_MODIFIED:
1349                printf("buffer modified after being freed\n");
1350                off = verify_pattern(KMEM_FREE_PATTERN, buf, cp->cache_verify);
1351                if (off == NULL)        /* shouldn't happen */
1352                        off = buf;
1353                printf("modification occurred at offset 0x%lx "
1354                    "(0x%llx replaced by 0x%llx)\n",
1355                    (uintptr_t)off - (uintptr_t)buf,
1356                    (longlong_t)KMEM_FREE_PATTERN, (longlong_t)*off);
1357                break;

1359        case KMERR_REDZONE:
1360                printf("redzone violation: write past end of buffer\n");
1361                break;

1363        case KMERR_BADADDR:
1364                printf("invalid free: buffer not in cache\n");
1365                break;

1367        case KMERR_DUPFREE:
1368                printf("duplicate free: buffer freed twice\n");
1369                break;

1371        case KMERR_BADBUFTAG:
1372                printf("boundary tag corrupted\n");
1373                printf("bcp ^ bxstat = %lx, should be %lx\n",
1374                    (intptr_t)btp->bt_bufctl ^ btp->bt_bxstat,
1375                    KMEM_BUFTAG_FREE);
1376                break;
```



```
1378        case KMERR_BADBUFCTL:
1379                printf("bufctl corrupted\n");
1380                break;

1382        case KMERR_BADCACHE:
1383                printf("buffer freed to wrong cache\n");
1384                printf("buffer was allocated from %s,\n", cp->cache_name);
1385                printf("caller attempting free to %s.\n", cparg->cache_name);
1386                break;

1388        case KMERR_BADSIZE:
1389                printf("bad free: free size (%u) != alloc size (%u)\n",
1390                    KMEM_SIZE_DECODE(((uint32_t *)btp)[0]),
1391                    KMEM_SIZE_DECODE(((uint32_t *)btp)[1]));
1392                break;

1394        case KMERR_BADBASE:
1395                printf("bad free: free address (%p) != alloc address (%p)\n",
1396                    bufarg, buf);
1397                break;
1398        }

1400        printf("buffer=%p  bufctl=%p  cache: %s\n",
1401            bufarg, (void *)bcp, cparg->cache_name);

1403        if (bcp != NULL && (cp->cache_flags & KMF_AUDIT) &&
1404            error != KMERR_BADBUFCTL) {
1405                int d;
1406                timestruc_t ts;
1407                kmem_bufctl_audit_t *bcap = (kmem_bufctl_audit_t *)bcp;

1409                hrt2ts(kmem_panic_info.kmp_timestamp - bcap->bc_timestamp, &ts);
1410                printf("previous transaction on buffer %p:\n", buf);
1411                printf("thread=%p  time=T-%ld.%09ld  slab=%p  cache: %s\n",
1412                    (void *)bcap->bc_thread, ts.tv_sec, ts.tv_nsec,
1413                    (void *)sp, cp->cache_name);
1414                for (d = 0; d < MIN(bcap->bc_depth, KMEM_STACK_DEPTH); d++) {
1415                        ulong_t off;
1416                        char *sym = kobj_getsymname(bcap->bc_stack[d], &off);
1417                        printf("%s+%lx\n", sym ? sym : "?", off);
1418                }
1419        }
1420        if (kmem_panic > 0)
1421                panic("kernel heap corruption detected");
1422        if (kmem_panic == 0)
1423                debug_enter(NULL);
1424        kmem_logging = 1;       /* resume logging */
1425 }

1427 static kmem_log_header_t *
1428 kmem_log_init(size_t logsize)
1429 {
1430        kmem_log_header_t *lhp;
1431        int nchunks = 4 * max_ncpus;
1432        size_t lhsize = (size_t)&((kmem_log_header_t *)0)->lh_cpu[max_ncpus];
1433        int i;

1435        /*
1436         * Make sure that lhp->lh_cpu[] is nicely aligned
1437         * to prevent false sharing of cache lines.
1438         */
1439        lhsize = P2ROUNDUP(lhsize, KMEM_ALIGN);
1440        lhp = vmem_xalloc(kmem_log_arena, lhsize, 64, P2NPHASE(lhsize, 64), 0,
1441            NULL, NULL, VM_SLEEP);
1442        bzero(lhp, lhsize);
```

```
1444         mutex_init(&lhp->lh_lock, NULL, MUTEX_DEFAULT, NULL);
1445         lhp->lh_nchunks = nchunks;
1446         lhp->lh_chunksize = P2ROUNDUP(logsize / nchunks + 1, PAGESIZE);
1447         lhp->lh_base = vmem_alloc(kmem_log_arena,
1448             lhp->lh_chunksize * nchunks, VM_SLEEP);
1449         lhp->lh_free = vmem_alloc(kmem_log_arena,
1450             nchunks * sizeof (int), VM_SLEEP);
1451         bzero(lhp->lh_base, lhp->lh_chunksize * nchunks);

1453         for (i = 0; i < max_ncpus; i++) {
1454                 kmem_cpu_log_header_t *clhp = &lhp->lh_cpu[i];
1455                 mutex_init(&clhp->clh_lock, NULL, MUTEX_DEFAULT, NULL);
1456                 clhp->clh_chunk = i;
1457         }

1459         for (i = max_ncpus; i < nchunks; i++)
1460                 lhp->lh_free[i] = i;

1462         lhp->lh_head = max_ncpus;
1463         lhp->lh_tail = 0;

1465         return (lhp);
1466 }

1468 static void *
1469 kmem_log_enter(kmem_log_header_t *lhp, void *data, size_t size)
1470 {
1471         void *logspace;
1472         kmem_cpu_log_header_t *clhp = &lhp->lh_cpu[CPU->cpu_seqid];

1474         if (lhp == NULL || kmem_logging == 0 || panicstr)
1475                 return (NULL);

1477         mutex_enter(&clhp->clh_lock);
1478         clhp->clh_hits++;
1479         if (size > clhp->clh_avail) {
1480                 mutex_enter(&lhp->lh_lock);
1481                 lhp->lh_hits++;
1482                 lhp->lh_free[lhp->lh_tail] = clhp->clh_chunk;
1483                 lhp->lh_tail = (lhp->lh_tail + 1) % lhp->lh_nchunks;
1484                 clhp->clh_chunk = lhp->lh_free[lhp->lh_head];
1485                 lhp->lh_head = (lhp->lh_head + 1) % lhp->lh_nchunks;
1486                 clhp->clh_current = lhp->lh_base +
1487                     clhp->clh_chunk * lhp->lh_chunksize;
1488                 clhp->clh_avail = lhp->lh_chunksize;
1489                 if (size > lhp->lh_chunksize)
1490                         size = lhp->lh_chunksize;
1491                 mutex_exit(&lhp->lh_lock);
1492         }
1493         logspace = clhp->clh_current;
1494         clhp->clh_current += size;
1495         clhp->clh_avail -= size;
1496         bcopy(data, logspace, size);
1497         mutex_exit(&clhp->clh_lock);
1498         return (logspace);
1499 }

1501 #define KMEM_AUDIT(lp, cp, bcp)                                         \
1502 {                                                                       \
1503         kmem_bufctl_audit_t *_bcp = (kmem_bufctl_audit_t *)(bcp);       \
1504         _bcp->bc_timestamp = gethrtime();                               \
1505         _bcp->bc_thread = curthread;                                    \
1506         _bcp->bc_depth = getpcstack(_bcp->bc_stack, KMEM_STACK_DEPTH);  \
1507         _bcp->bc_lastlog = kmem_log_enter((lp), _bcp, sizeof (*_bcp));  \
1508 }
```



```
1510 static void
1511 kmem_log_event(kmem_log_header_t *lp, kmem_cache_t *cp,
1512         kmem_slab_t *sp, void *addr)
1513 {
1514         kmem_bufctl_audit_t bca;

1516         bzero(&bca, sizeof (kmem_bufctl_audit_t));
1517         bca.bc_addr = addr;
1518         bca.bc_slab = sp;
1519         bca.bc_cache = cp;
1520         KMEM_AUDIT(lp, cp, &bca);
1521 }

1523 /*
1524  * Create a new slab for cache cp.
1525  */
1526 static kmem_slab_t *
1527 kmem_slab_create(kmem_cache_t *cp, int kmflag)
1528 {
1529         size_t slabsize = cp->cache_slabsize;
1530         size_t chunksize = cp->cache_chunksize;
1531         int cache_flags = cp->cache_flags;
1532         size_t color, chunks;
1533         char *buf, *slab;
1534         kmem_slab_t *sp;
1535         kmem_bufctl_t *bcp;
1536         vmem_t *vmp = cp->cache_arena;

1538         ASSERT(MUTEX_NOT_HELD(&cp->cache_lock));

1540         color = cp->cache_color + cp->cache_align;
1541         if (color > cp->cache_maxcolor)
1542                 color = cp->cache_mincolor;
1543         cp->cache_color = color;

1545         slab = vmem_alloc(vmp, slabsize, kmflag & KM_VMFLAGS);

1547         if (slab == NULL)
1548                 goto vmem_alloc_failure;

1550         ASSERT(P2PHASE((uintptr_t)slab, vmp->vm_quantum) == 0);

1552         /*
1553          * Reverify what was already checked in kmem_cache_set_move(), since the
1554          * consolidator depends (for correctness) on slabs being initialized
1555          * with the 0xbaddcafe memory pattern (setting a low order bit usable by
1556          * clients to distinguish uninitialized memory from known objects).
1557          */
1558         ASSERT((cp->cache_move == NULL) || !(cp->cache_cflags & KMC_NOTOUCH));
1559         if (!(cp->cache_cflags & KMC_NOTOUCH))
1560                 copy_pattern(KMEM_UNINITIALIZED_PATTERN, slab, slabsize);

1562         if (cache_flags & KMF_HASH) {
1563                 if ((sp = kmem_cache_alloc(kmem_slab_cache, kmflag)) == NULL)
1564                         goto slab_alloc_failure;
1565                 chunks = (slabsize - color) / chunksize;
1566         } else {
1567                 sp = KMEM_SLAB(cp, slab);
1568                 chunks = (slabsize - sizeof (kmem_slab_t) - color) / chunksize;
1569         }

1571         sp->slab_cache  = cp;
1572         sp->slab_head   = NULL;
1573         sp->slab_refcnt = 0;
1574         sp->slab_base   = buf = slab + color;
1575         sp->slab_chunks = chunks;
```

```
1576         sp->slab_stuck_offset = (uint32_t)-1;
1577         sp->slab_later_count = 0;
1578         sp->slab_flags = 0;

1580         ASSERT(chunks > 0);
1581         while (chunks-- != 0) {
1582                 if (cache_flags & KMF_HASH) {
1583                         bcp = kmem_cache_alloc(cp->cache_bufctl_cache, kmflag);
1584                         if (bcp == NULL)
1585                                 goto bufctl_alloc_failure;
1586                         if (cache_flags & KMF_AUDIT) {
1587                                 kmem_bufctl_audit_t *bcap =
1588                                     (kmem_bufctl_audit_t *)bcp;
1589                                 bzero(bcap, sizeof (kmem_bufctl_audit_t));
1590                                 bcap->bc_cache = cp;
1591                         }
1592                         bcp->bc_addr = buf;
1593                         bcp->bc_slab = sp;
1594                 } else {
1595                         bcp = KMEM_BUFCTL(cp, buf);
1596                 }
1597                 if (cache_flags & KMF_BUFTAG) {
1598                         kmem_buftag_t *btp = KMEM_BUFTAG(cp, buf);
1599                         btp->bt_redzone = KMEM_REDZONE_PATTERN;
1600                         btp->bt_bufctl = bcp;
1601                         btp->bt_bxstat = (intptr_t)bcp ^ KMEM_BUFTAG_FREE;
1602                         if (cache_flags & KMF_DEADBEEF) {
1603                                 copy_pattern(KMEM_FREE_PATTERN, buf,
1604                                     cp->cache_verify);
1605                         }
1606                 }
1607                 bcp->bc_next = sp->slab_head;
1608                 sp->slab_head = bcp;
1609                 buf += chunksize;
1610         }

1612         kmem_log_event(kmem_slab_log, cp, sp, slab);

1614         return (sp);

1616 bufctl_alloc_failure:

1618         while ((bcp = sp->slab_head) != NULL) {
1619                 sp->slab_head = bcp->bc_next;
1620                 kmem_cache_free(cp->cache_bufctl_cache, bcp);
1621         }
1622         kmem_cache_free(kmem_slab_cache, sp);

1624 slab_alloc_failure:

1626         vmem_free(vmp, slab, slabsize);

1628 vmem_alloc_failure:

1630         kmem_log_event(kmem_failure_log, cp, NULL, NULL);
1631         atomic_inc_64(&cp->cache_alloc_fail);

1633         return (NULL);
1634 }

1636 /*
1637  * Destroy a slab.
1638  */
1639 static void
1640 kmem_slab_destroy(kmem_cache_t *cp, kmem_slab_t *sp)
1641 {
```



```
1642         vmem_t *vmp = cp->cache_arena;
1643         void *slab = (void *)P2ALIGN((uintptr_t)sp->slab_base, vmp->vm_quantum);

1645         ASSERT(MUTEX_NOT_HELD(&cp->cache_lock));
1646         ASSERT(sp->slab_refcnt == 0);

1648         if (cp->cache_flags & KMF_HASH) {
1649                 kmem_bufctl_t *bcp;
1650                 while ((bcp = sp->slab_head) != NULL) {
1651                         sp->slab_head = bcp->bc_next;
1652                         kmem_cache_free(cp->cache_bufctl_cache, bcp);
1653                 }
1654                 kmem_cache_free(kmem_slab_cache, sp);
1655         }
1656         vmem_free(vmp, slab, cp->cache_slabsize);
1657 }

1659 static void *
1660 kmem_slab_alloc_impl(kmem_cache_t *cp, kmem_slab_t *sp, boolean_t prefill)
1661 {
1662         kmem_bufctl_t *bcp, **hash_bucket;
1663         void *buf;
1664         boolean_t new_slab = (sp->slab_refcnt == 0);

1666         ASSERT(MUTEX_HELD(&cp->cache_lock));
1667         /*
1668          * kmem_slab_alloc() drops cache_lock when it creates a new slab, so we
1669          * can't ASSERT(avl_is_empty(&cp->cache_partial_slabs)) here when the
1670          * slab is newly created.
1671          */
1672         ASSERT(new_slab || (KMEM_SLAB_IS_PARTIAL(sp) &&
1673             (sp == avl_first(&cp->cache_partial_slabs))));
1674         ASSERT(sp->slab_cache == cp);

1676         cp->cache_slab_alloc++;
1677         cp->cache_bufslab--;
1678         sp->slab_refcnt++;

1680         bcp = sp->slab_head;
1681         sp->slab_head = bcp->bc_next;

1683         if (cp->cache_flags & KMF_HASH) {
1684                 /*
1685                  * Add buffer to allocated-address hash table.
1686                  */
1687                 buf = bcp->bc_addr;
1688                 hash_bucket = KMEM_HASH(cp, buf);
1689                 bcp->bc_next = *hash_bucket;
1690                 *hash_bucket = bcp;
1691                 if ((cp->cache_flags & (KMF_AUDIT | KMF_BUFTAG)) == KMF_AUDIT) {
1692                         KMEM_AUDIT(kmem_transaction_log, cp, bcp);
1693                 }
1694         } else {
1695                 buf = KMEM_BUF(cp, bcp);
1696         }

1698         ASSERT(KMEM_SLAB_MEMBER(sp, buf));

1700         if (sp->slab_head == NULL) {
1701                 ASSERT(KMEM_SLAB_IS_ALL_USED(sp));
1702                 if (new_slab) {
1703                         ASSERT(sp->slab_chunks == 1);
1704                 } else {
1705                         ASSERT(sp->slab_chunks > 1); /* the slab was partial */
1706                         avl_remove(&cp->cache_partial_slabs, sp);
1707                         sp->slab_later_count = 0; /* clear history */
```

```
1708                        sp->slab_flags &= ~KMEM_SLAB_NOMOVE;
1709                        sp->slab_stuck_offset = (uint32_t)-1;
1710                }
1711                list_insert_head(&cp->cache_complete_slabs, sp);
1712                cp->cache_complete_slab_count++;
1713                return (buf);
1714        }

1716        ASSERT(KMEM_SLAB_IS_PARTIAL(sp));
1717        /*
1718         * Peek to see if the magazine layer is enabled before
1719         * we prefill.  We're not holding the cpu cache lock,
1720         * so the peek could be wrong, but there's no harm in it.
1721         */
1722        if (new_slab && prefill && (cp->cache_flags & KMF_PREFILL) &&
1723            (KMEM_CPU_CACHE(cp)->cc_magsize != 0))  {
1724                kmem_slab_prefill(cp, sp);
1725                return (buf);
1726        }

1728        if (new_slab) {
1729                avl_add(&cp->cache_partial_slabs, sp);
1730                return (buf);
1731        }

1733        /*
1734         * The slab is now more allocated than it was, so the
1735         * order remains unchanged.
1736         */
1737        ASSERT(!avl_update(&cp->cache_partial_slabs, sp));
1738        return (buf);
1739 }

1741 /*
1742  * Allocate a raw (unconstructed) buffer from cp's slab layer.
1743  */
1744 static void *
1745 kmem_slab_alloc(kmem_cache_t *cp, int kmflag)
1746 {
1747        kmem_slab_t *sp;
1748        void *buf;
1749        boolean_t test_destructor;

1751        mutex_enter(&cp->cache_lock);
1752        test_destructor = (cp->cache_slab_alloc == 0);
1753        sp = avl_first(&cp->cache_partial_slabs);
1754        if (sp == NULL) {
1755                ASSERT(cp->cache_bufslab == 0);

1757                /*
1758                 * The freelist is empty.  Create a new slab.
1759                 */
1760                mutex_exit(&cp->cache_lock);
1761                if ((sp = kmem_slab_create(cp, kmflag)) == NULL) {
1762                        return (NULL);
1763                }
1764                mutex_enter(&cp->cache_lock);
1765                cp->cache_slab_create++;
1766                if ((cp->cache_buftotal += sp->slab_chunks) > cp->cache_bufmax)
1767                        cp->cache_bufmax = cp->cache_buftotal;
1768                cp->cache_bufslab += sp->slab_chunks;
1769        }

1771        buf = kmem_slab_alloc_impl(cp, sp, B_TRUE);
1772        ASSERT((cp->cache_slab_create - cp->cache_slab_destroy) ==
1773            (cp->cache_complete_slab_count +
```



```
1774            avl_numnodes(&cp->cache_partial_slabs) +
1775            (cp->cache_defrag == NULL ? 0 : cp->cache_defrag->kmd_deadcount)));
1776        mutex_exit(&cp->cache_lock);

1778        if (test_destructor && cp->cache_destructor != NULL) {
1779                /*
1780                 * On the first kmem_slab_alloc(), assert that it is valid to
1781                 * call the destructor on a newly constructed object without any
1782                 * client involvement.
1783                 */
1784                if ((cp->cache_constructor == NULL) ||
1785                    cp->cache_constructor(buf, cp->cache_private,
1786                    kmflag) == 0) {
1787                        cp->cache_destructor(buf, cp->cache_private);
1788                }
1789                copy_pattern(KMEM_UNINITIALIZED_PATTERN, buf,
1790                    cp->cache_bufsize);
1791                if (cp->cache_flags & KMF_DEADBEEF) {
1792                        copy_pattern(KMEM_FREE_PATTERN, buf, cp->cache_verify);
1793                }
1794        }

1796        return (buf);
1797 }

1799 static void kmem_slab_move_yes(kmem_cache_t *, kmem_slab_t *, void *);

1801 /*
1802  * Free a raw (unconstructed) buffer to cp's slab layer.
1803  */
1804 static void
1805 kmem_slab_free(kmem_cache_t *cp, void *buf)
1806 {
1807        kmem_slab_t *sp;
1808        kmem_bufctl_t *bcp, **prev_bcpp;

1810        ASSERT(buf != NULL);

1812        mutex_enter(&cp->cache_lock);
1813        cp->cache_slab_free++;

1815        if (cp->cache_flags & KMF_HASH) {
1816                /*
1817                 * Look up buffer in allocated-address hash table.
1818                 */
1819                prev_bcpp = KMEM_HASH(cp, buf);
1820                while ((bcp = *prev_bcpp) != NULL) {
1821                        if (bcp->bc_addr == buf) {
1822                                *prev_bcpp = bcp->bc_next;
1823                                sp = bcp->bc_slab;
1824                                break;
1825                        }
1826                        cp->cache_lookup_depth++;
1827                        prev_bcpp = &bcp->bc_next;
1828                }
1829        } else {
1830                bcp = KMEM_BUFCTL(cp, buf);
1831                sp = KMEM_SLAB(cp, buf);
1832        }

1834        if (bcp == NULL || sp->slab_cache != cp || !KMEM_SLAB_MEMBER(sp, buf)) {
1835                mutex_exit(&cp->cache_lock);
1836                kmem_error(KMERR_BADADDR, cp, buf);
1837                return;
1838        }
```

```
1840         if (KMEM_SLAB_OFFSET(sp, buf) == sp->slab_stuck_offset) {
1841                 /*
1842                  * If this is the buffer that prevented the consolidator from
1843                  * clearing the slab, we can reset the slab flags now that the
1844                  * buffer is freed. (It makes sense to do this in
1845                  * kmem_cache_free(), where the client gives up ownership of the
1846                  * buffer, but on the hot path the test is too expensive.)
1847                  */
1848                 kmem_slab_move_yes(cp, sp, buf);
1849         }
1850
1851         if ((cp->cache_flags & (KMF_AUDIT | KMF_BUFTAG)) == KMF_AUDIT) {
1852                 if (cp->cache_flags & KMF_CONTENTS)
1853                         ((kmem_bufctl_audit_t *)bcp)->bc_contents =
1854                             kmem_log_enter(kmem_content_log, buf,
1855                             cp->cache_contents);
1856                 KMEM_AUDIT(kmem_transaction_log, cp, bcp);
1857         }
1858
1859         bcp->bc_next = sp->slab_head;
1860         sp->slab_head = bcp;
1861
1862         cp->cache_bufslab++;
1863         ASSERT(sp->slab_refcnt >= 1);
1864
1865         if (--sp->slab_refcnt == 0) {
1866                 /*
1867                  * There are no outstanding allocations from this slab,
1868                  * so we can reclaim the memory.
1869                  */
1870                 if (sp->slab_chunks == 1) {
1871                         list_remove(&cp->cache_complete_slabs, sp);
1872                         cp->cache_complete_slab_count--;
1873                 } else {
1874                         avl_remove(&cp->cache_partial_slabs, sp);
1875                 }
1876
1877                 cp->cache_buftotal -= sp->slab_chunks;
1878                 cp->cache_bufslab -= sp->slab_chunks;
1879                 /*
1880                  * Defer releasing the slab to the virtual memory subsystem
1881                  * while there is a pending move callback, since we guarantee
1882                  * that buffers passed to the move callback have only been
1883                  * touched by kmem or by the client itself. Since the memory
1884                  * patterns baddcafe (uninitialized) and deadbeef (freed) both
1885                  * set at least one of the two lowest order bits, the client can
1886                  * test those bits in the move callback to determine whether or
1887                  * not it knows about the buffer (assuming that the client also
1888                  * sets one of those low order bits whenever it frees a buffer).
1889                  */
1890                 if (cp->cache_defrag == NULL ||
1891                     (avl_is_empty(&cp->cache_defrag->kmd_moves_pending) &&
1892                     !(sp->slab_flags & KMEM_SLAB_MOVE_PENDING))) {
1893                         cp->cache_slab_destroy++;
1894                         mutex_exit(&cp->cache_lock);
1895                         kmem_slab_destroy(cp, sp);
1896                 } else {
1897                         list_t *deadlist = &cp->cache_defrag->kmd_deadlist;
1898                         /*
1899                          * Slabs are inserted at both ends of the deadlist to
1900                          * distinguish between slabs freed while move callbacks
1901                          * are pending (list head) and a slab freed while the
1902                          * lock is dropped in kmem_move_buffers() (list tail) so
1903                          * that in both cases slab_destroy() is called from the
1904                          * right context.
1905                          */
```



```
1906                         if (sp->slab_flags & KMEM_SLAB_MOVE_PENDING) {
1907                                 list_insert_tail(deadlist, sp);
1908                         } else {
1909                                 list_insert_head(deadlist, sp);
1910                         }
1911                         cp->cache_defrag->kmd_deadcount++;
1912                         mutex_exit(&cp->cache_lock);
1913                 }
1914                 return;
1915         }
1916
1917         if (bcp->bc_next == NULL) {
1918                 /* Transition the slab from completely allocated to partial. */
1919                 ASSERT(sp->slab_refcnt == (sp->slab_chunks - 1));
1920                 ASSERT(sp->slab_chunks > 1);
1921                 list_remove(&cp->cache_complete_slabs, sp);
1922                 cp->cache_complete_slab_count--;
1923                 avl_add(&cp->cache_partial_slabs, sp);
1924         } else {
1925 #ifdef  DEBUG
1926                 if (avl_update_gt(&cp->cache_partial_slabs, sp)) {
1927                         KMEM_STAT_ADD(kmem_move_stats.kms_avl_update);
1928                 } else {
1929                         KMEM_STAT_ADD(kmem_move_stats.kms_avl_noupdate);
1930                 }
1931 #else
1932                 (void) avl_update_gt(&cp->cache_partial_slabs, sp);
1933 #endif
1934         }
1935
1936         ASSERT((cp->cache_slab_create - cp->cache_slab_destroy) ==
1937             (cp->cache_complete_slab_count +
1938             avl_numnodes(&cp->cache_partial_slabs) +
1939             (cp->cache_defrag == NULL ? 0 : cp->cache_defrag->kmd_deadcount)));
1940         mutex_exit(&cp->cache_lock);
1941 }
1942
1943 /*
1944  * Return -1 if kmem_error, 1 if constructor fails, 0 if successful.
1945  */
1946 static int
1947 kmem_cache_alloc_debug(kmem_cache_t *cp, void *buf, int kmflag, int construct,
1948     caddr_t caller)
1949 {
1950         kmem_buftag_t *btp = KMEM_BUFTAG(cp, buf);
1951         kmem_bufctl_audit_t *bcp = (kmem_bufctl_audit_t *)btp->bt_bufctl;
1952         uint32_t mtbf;
1953
1954         if (btp->bt_bxstat != ((intptr_t)bcp ^ KMEM_BUFTAG_FREE)) {
1955                 kmem_error(KMERR_BADBUFTAG, cp, buf);
1956                 return (-1);
1957         }
1958
1959         btp->bt_bxstat = (intptr_t)bcp ^ KMEM_BUFTAG_ALLOC;
1960
1961         if ((cp->cache_flags & KMF_HASH) && bcp->bc_addr != buf) {
1962                 kmem_error(KMERR_BADBUFCTL, cp, buf);
1963                 return (-1);
1964         }
1965
1966         if (cp->cache_flags & KMF_DEADBEEF) {
1967                 if (!construct && (cp->cache_flags & KMF_LITE)) {
1968                         if (*(uint64_t *)buf != KMEM_FREE_PATTERN) {
1969                                 kmem_error(KMERR_MODIFIED, cp, buf);
1970                                 return (-1);
1971                         }
```

```
1972                        if (cp->cache_constructor != NULL)
1973                                *(uint64_t *)buf = btp->bt_redzone;
1974                        else
1975                                *(uint64_t *)buf = KMEM_UNINITIALIZED_PATTERN;
1976                } else {
1977                        construct = 1;
1978                        if (verify_and_copy_pattern(KMEM_FREE_PATTERN,
1979                            KMEM_UNINITIALIZED_PATTERN, buf,
1980                            cp->cache_verify)) {
1981                                kmem_error(KMERR_MODIFIED, cp, buf);
1982                                return (-1);
1983                        }
1984                }
1985        }
1986        btp->bt_redzone = KMEM_REDZONE_PATTERN;

1988        if ((mtbf = kmem_mtbf | cp->cache_mtbf) != 0 &&
1989            gethrtime() % mtbf == 0 &&
1990            (kmflag & (KM_NOSLEEP | KM_PANIC)) == KM_NOSLEEP) {
1991                kmem_log_event(kmem_failure_log, cp, NULL, NULL);
1992                if (!construct && cp->cache_destructor != NULL)
1993                        cp->cache_destructor(buf, cp->cache_private);
1994        } else {
1995                mtbf = 0;
1996        }

1998        if (mtbf || (construct && cp->cache_constructor != NULL &&
1999            cp->cache_constructor(buf, cp->cache_private, kmflag) != 0)) {
2000                atomic_inc_64(&cp->cache_alloc_fail);
2001                btp->bt_bxstat = (intptr_t)bcp ^ KMEM_BUFTAG_FREE;
2002                if (cp->cache_flags & KMF_DEADBEEF)
2003                        copy_pattern(KMEM_FREE_PATTERN, buf, cp->cache_verify);
2004                kmem_slab_free(cp, buf);
2005                return (1);
2006        }

2008        if (cp->cache_flags & KMF_AUDIT) {
2009                KMEM_AUDIT(kmem_transaction_log, cp, bcp);
2010        }

2012        if ((cp->cache_flags & KMF_LITE) &&
2013            !(cp->cache_cflags & KMC_KMEM_ALLOC)) {
2014                KMEM_BUFTAG_LITE_ENTER(btp, kmem_lite_count, caller);
2015        }

2017        return (0);
2018 }

2020 static int
2021 kmem_cache_free_debug(kmem_cache_t *cp, void *buf, caddr_t caller)
2022 {
2023        kmem_buftag_t *btp = KMEM_BUFTAG(cp, buf);
2024        kmem_bufctl_audit_t *bcp = (kmem_bufctl_audit_t *)btp->bt_bufctl;
2025        kmem_slab_t *sp;

2027        if (btp->bt_bxstat != ((intptr_t)bcp ^ KMEM_BUFTAG_ALLOC)) {
2028                if (btp->bt_bxstat == ((intptr_t)bcp ^ KMEM_BUFTAG_FREE)) {
2029                        kmem_error(KMERR_DUPFREE, cp, buf);
2030                        return (-1);
2031                }
2032                sp = kmem_findslab(cp, buf);
2033                if (sp == NULL || sp->slab_cache != cp)
2034                        kmem_error(KMERR_BADADDR, cp, buf);
2035                else
2036                        kmem_error(KMERR_REDZONE, cp, buf);
2037                return (-1);
```



```
2038        }

2040        btp->bt_bxstat = (intptr_t)bcp ^ KMEM_BUFTAG_FREE;

2042        if ((cp->cache_flags & KMF_HASH) && bcp->bc_addr != buf) {
2043                kmem_error(KMERR_BADBUFCTL, cp, buf);
2044                return (-1);
2045        }

2047        if (btp->bt_redzone != KMEM_REDZONE_PATTERN) {
2048                kmem_error(KMERR_REDZONE, cp, buf);
2049                return (-1);
2050        }

2052        if (cp->cache_flags & KMF_AUDIT) {
2053                if (cp->cache_flags & KMF_CONTENTS)
2054                        bcp->bc_contents = kmem_log_enter(kmem_content_log,
2055                            buf, cp->cache_contents);
2056                KMEM_AUDIT(kmem_transaction_log, cp, bcp);
2057        }

2059        if ((cp->cache_flags & KMF_LITE) &&
2060            !(cp->cache_cflags & KMC_KMEM_ALLOC)) {
2061                KMEM_BUFTAG_LITE_ENTER(btp, kmem_lite_count, caller);
2062        }

2064        if (cp->cache_flags & KMF_DEADBEEF) {
2065                if (cp->cache_flags & KMF_LITE)
2066                        btp->bt_redzone = *(uint64_t *)buf;
2067                else if (cp->cache_destructor != NULL)
2068                        cp->cache_destructor(buf, cp->cache_private);

2070                copy_pattern(KMEM_FREE_PATTERN, buf, cp->cache_verify);
2071        }

2073        return (0);
2074 }

2076 /*
2077  * Free each object in magazine mp to cp's slab layer, and free mp itself.
2078  */
2079 static void
2080 kmem_magazine_destroy(kmem_cache_t *cp, kmem_magazine_t *mp, int nrounds)
2081 {
2082        int round;

2084        ASSERT(!list_link_active(&cp->cache_link) ||
2085            taskq_member(kmem_taskq, curthread));

2087        for (round = 0; round < nrounds; round++) {
2088                void *buf = mp->mag_round[round];

2090                if (cp->cache_flags & KMF_DEADBEEF) {
2091                        if (verify_pattern(KMEM_FREE_PATTERN, buf,
2092                            cp->cache_verify) != NULL) {
2093                                kmem_error(KMERR_MODIFIED, cp, buf);
2094                                continue;
2095                        }
2096                        if ((cp->cache_flags & KMF_LITE) &&
2097                            cp->cache_destructor != NULL) {
2098                                kmem_buftag_t *btp = KMEM_BUFTAG(cp, buf);
2099                                *(uint64_t *)buf = btp->bt_redzone;
2100                                cp->cache_destructor(buf, cp->cache_private);
2101                                *(uint64_t *)buf = KMEM_FREE_PATTERN;
2102                        }
2103                } else if (cp->cache_destructor != NULL) {
```

```
2104                         cp->cache_destructor(buf, cp->cache_private);
2105                 }
2106
2107                 kmem_slab_free(cp, buf);
2108         }
2109         ASSERT(KMEM_MAGAZINE_VALID(cp, mp));
2110         kmem_cache_free(cp->cache_magtype->mt_cache, mp);
2111 }
2112
2113 /*
2114  * Allocate a magazine from the depot.
2115  */
2116 static kmem_magazine_t *
2117 kmem_depot_alloc(kmem_cache_t *cp, kmem_maglist_t *mlp)
2118 {
2119         kmem_magazine_t *mp;
2120
2121         /*
2122          * If we can't get the depot lock without contention,
2123          * update our contention count.  We use the depot
2124          * contention rate to determine whether we need to
2125          * increase the magazine size for better scalability.
2126          */
2127         if (!mutex_tryenter(&cp->cache_depot_lock)) {
2128                 mutex_enter(&cp->cache_depot_lock);
2129                 cp->cache_depot_contention++;
2130         }
2131
2132         if ((mp = mlp->ml_list) != NULL) {
2133                 ASSERT(KMEM_MAGAZINE_VALID(cp, mp));
2134                 mlp->ml_list = mp->mag_next;
2135                 if (--mlp->ml_total < mlp->ml_min)
2136                         mlp->ml_min = mlp->ml_total;
2137                 mlp->ml_alloc++;
2138         }
2139
2140         mutex_exit(&cp->cache_depot_lock);
2141
2142         return (mp);
2143 }
2144
2145 /*
2146  * Free a magazine to the depot.
2147  */
2148 static void
2149 kmem_depot_free(kmem_cache_t *cp, kmem_maglist_t *mlp, kmem_magazine_t *mp)
2150 {
2151         mutex_enter(&cp->cache_depot_lock);
2152         ASSERT(KMEM_MAGAZINE_VALID(cp, mp));
2153         mp->mag_next = mlp->ml_list;
2154         mlp->ml_list = mp;
2155         mlp->ml_total++;
2156         mutex_exit(&cp->cache_depot_lock);
2157 }
2158
2159 /*
2160  * Update the working set statistics for cp's depot.
2161  */
2162 static void
2163 kmem_depot_ws_update(kmem_cache_t *cp)
2164 {
2165         mutex_enter(&cp->cache_depot_lock);
2166         cp->cache_full.ml_reaplimit = cp->cache_full.ml_min;
2167         cp->cache_full.ml_min = cp->cache_full.ml_total;
2168         cp->cache_empty.ml_reaplimit = cp->cache_empty.ml_min;
2169         cp->cache_empty.ml_min = cp->cache_empty.ml_total;
```



```
2170         mutex_exit(&cp->cache_depot_lock);
2171 }
2172
2173 /*
2174  * Set the working set statistics for cp's depot to zero.  (Everything is
2175  * eligible for reaping.)
2176  */
2177 static void
2178 kmem_depot_ws_zero(kmem_cache_t *cp)
2179 {
2180         mutex_enter(&cp->cache_depot_lock);
2181         cp->cache_full.ml_reaplimit = cp->cache_full.ml_total;
2182         cp->cache_full.ml_min = cp->cache_full.ml_total;
2183         cp->cache_empty.ml_reaplimit = cp->cache_empty.ml_total;
2184         cp->cache_empty.ml_min = cp->cache_empty.ml_total;
2185         mutex_exit(&cp->cache_depot_lock);
2186 }
2187
2188 /*
2189 #endif /* ! codereview */
2190  * Reap all magazines that have fallen out of the depot's working set.
2191  */
2192 static void
2193 kmem_depot_ws_reap(kmem_cache_t *cp)
2194 {
2195         long reap;
2196         kmem_magazine_t *mp;
2197
2198         ASSERT(!list_link_active(&cp->cache_link) ||
2199             taskq_member(kmem_taskq, curthread));
2200
2201         reap = MIN(cp->cache_full.ml_reaplimit, cp->cache_full.ml_min);
2202         while (reap-- && (mp = kmem_depot_alloc(cp, &cp->cache_full)) != NULL)
2203                 kmem_magazine_destroy(cp, mp, cp->cache_magtype->mt_magsize);
2204
2205         reap = MIN(cp->cache_empty.ml_reaplimit, cp->cache_empty.ml_min);
2206         while (reap-- && (mp = kmem_depot_alloc(cp, &cp->cache_empty)) != NULL)
2207                 kmem_magazine_destroy(cp, mp, 0);
2208 }
2209
2210 static void
2211 kmem_cpu_reload(kmem_cpu_cache_t *ccp, kmem_magazine_t *mp, int rounds)
2212 {
2213         ASSERT((ccp->cc_loaded == NULL && ccp->cc_rounds == -1) ||
2214             (ccp->cc_loaded && ccp->cc_rounds + rounds == ccp->cc_magsize));
2215         ASSERT(ccp->cc_magsize > 0);
2216
2217         ccp->cc_ploaded = ccp->cc_loaded;
2218         ccp->cc_prounds = ccp->cc_rounds;
2219         ccp->cc_loaded = mp;
2220         ccp->cc_rounds = rounds;
2221 }
2222
2223 /*
2224  * Intercept kmem alloc/free calls during crash dump in order to avoid
2225  * changing kmem state while memory is being saved to the dump device.
2226  * Otherwise, ::kmem_verify will report "corrupt buffers".  Note that
2227  * there are no locks because only one CPU calls kmem during a crash
2228  * dump. To enable this feature, first create the associated vmem
2229  * arena with VMC_DUMPSAFE.
2230  */
2231 static void *kmem_dump_start;   /* start of pre-reserved heap */
2232 static void *kmem_dump_end;     /* end of heap area */
2233 static void *kmem_dump_curr;    /* current free heap pointer */
2234 static size_t kmem_dump_size;   /* size of heap area */
```

```
2236 /* append to each buf created in the pre-reserved heap */
2237 typedef struct kmem_dumpctl {
2238         void    *kdc_next;      /* cache dump free list linkage */
2239 } kmem_dumpctl_t;

2241 #define KMEM_DUMPCTL(cp, buf)   \
2242         ((kmem_dumpctl_t *)P2ROUNDUP((uintptr_t)(buf) + (cp)->cache_bufsize, \
2243             sizeof (void *)))

2245 /* Keep some simple stats. */
2246 #define KMEM_DUMP_LOGS  (100)

2248 typedef struct kmem_dump_log {
2249         kmem_cache_t    *kdl_cache;
2250         uint_t          kdl_allocs;             /* # of dump allocations */
2251         uint_t          kdl_frees;              /* # of dump frees */
2252         uint_t          kdl_alloc_fails;        /* # of allocation failures */
2253         uint_t          kdl_free_nondump;       /* # of non-dump frees */
2254         uint_t          kdl_unsafe;             /* cache was used, but unsafe */
2255 } kmem_dump_log_t;

2257 static kmem_dump_log_t *kmem_dump_log;
2258 static int kmem_dump_log_idx;

2260 #define KDI_LOG(cp, stat) {                                             \
2261         kmem_dump_log_t *kdl;                                          \
2262         if ((kdl = (kmem_dump_log_t *)((cp)->cache_dumplog)) != NULL) { \
2263                 kdl->stat++;                                           \
2264         } else if (kmem_dump_log_idx < KMEM_DUMP_LOGS) {               \
2265                 kdl = &kmem_dump_log[kmem_dump_log_idx++];             \
2266                 kdl->stat++;                                           \
2267                 kdl->kdl_cache = (cp);                                 \
2268                 (cp)->cache_dumplog = kdl;                             \
2269         }                                                              \
2270 }

2272 /* set non zero for full report */
2273 uint_t kmem_dump_verbose = 0;

2275 /* stats for overize heap */
2276 uint_t kmem_dump_oversize_allocs = 0;
2277 uint_t kmem_dump_oversize_max = 0;

2279 static void
2280 kmem_dumppr(char **pp, char *e, const char *format, ...)
2281 {
2282         char *p = *pp;

2284         if (p < e) {
2285                 int n;
2286                 va_list ap;

2288                 va_start(ap, format);
2289                 n = vsnprintf(p, e - p, format, ap);
2290                 va_end(ap);
2291                 *pp = p + n;
2292         }
2293 }

2295 /*
2296  * Called when dumpadm(1M) configures dump parameters.
2297  */
2298 void
2299 kmem_dump_init(size_t size)
2300 {
2301         if (kmem_dump_start != NULL)
```



```
2302                 kmem_free(kmem_dump_start, kmem_dump_size);

2304         if (kmem_dump_log == NULL)
2305                 kmem_dump_log = (kmem_dump_log_t *)kmem_zalloc(KMEM_DUMP_LOGS *
2306                     sizeof (kmem_dump_log_t), KM_SLEEP);

2308         kmem_dump_start = kmem_alloc(size, KM_SLEEP);

2310         if (kmem_dump_start != NULL) {
2311                 kmem_dump_size = size;
2312                 kmem_dump_curr = kmem_dump_start;
2313                 kmem_dump_end = (void *)((char *)kmem_dump_start + size);
2314                 copy_pattern(KMEM_UNINITIALIZED_PATTERN, kmem_dump_start, size);
2315         } else {
2316                 kmem_dump_size = 0;
2317                 kmem_dump_curr = NULL;
2318                 kmem_dump_end = NULL;
2319         }
2320 }

2322 /*
2323  * Set flag for each kmem_cache_t if is safe to use alternate dump
2324  * memory. Called just before panic crash dump starts. Set the flag
2325  * for the calling CPU.
2326  */
2327 void
2328 kmem_dump_begin(void)
2329 {
2330         ASSERT(panicstr != NULL);
2331         if (kmem_dump_start != NULL) {
2332                 kmem_cache_t *cp;

2334                 for (cp = list_head(&kmem_caches); cp != NULL;
2335                     cp = list_next(&kmem_caches, cp)) {
2336                         kmem_cpu_cache_t *ccp = KMEM_CPU_CACHE(cp);

2338                         if (cp->cache_arena->vm_cflags & VMC_DUMPSAFE) {
2339                                 cp->cache_flags |= KMF_DUMPDIVERT;
2340                                 ccp->cc_flags |= KMF_DUMPDIVERT;
2341                                 ccp->cc_dump_rounds = ccp->cc_rounds;
2342                                 ccp->cc_dump_prounds = ccp->cc_prounds;
2343                                 ccp->cc_rounds = ccp->cc_prounds = -1;
2344                         } else {
2345                                 cp->cache_flags |= KMF_DUMPUNSAFE;
2346                                 ccp->cc_flags |= KMF_DUMPUNSAFE;
2347                         }
2348                 }
2349         }
2350 }

2352 /*
2353  * finished dump intercept
2354  * print any warnings on the console
2355  * return verbose information to dumpsys() in the given buffer
2356  */
2357 size_t
2358 kmem_dump_finish(char *buf, size_t size)
2359 {
2360         int kdi_idx;
2361         int kdi_end = kmem_dump_log_idx;
2362         int percent = 0;
2363         int header = 0;
2364         int warn = 0;
2365         size_t used;
2366         kmem_cache_t *cp;
2367         kmem_dump_log_t *kdl;
```

```
2368         char *e = buf + size;
2369         char *p = buf;

2371         if (kmem_dump_size == 0 || kmem_dump_verbose == 0)
2372                 return (0);

2374         used = (char *)kmem_dump_curr - (char *)kmem_dump_start;
2375         percent = (used * 100) / kmem_dump_size;

2377         kmem_dumppr(&p, e, "%% heap used,%d\n", percent);
2378         kmem_dumppr(&p, e, "used bytes,%ld\n", used);
2379         kmem_dumppr(&p, e, "heap size,%ld\n", kmem_dump_size);
2380         kmem_dumppr(&p, e, "Oversize allocs,%d\n",
2381             kmem_dump_oversize_allocs);
2382         kmem_dumppr(&p, e, "Oversize max size,%ld\n",
2383             kmem_dump_oversize_max);

2385         for (kdi_idx = 0; kdi_idx < kdi_end; kdi_idx++) {
2386                 kdl = &kmem_dump_log[kdi_idx];
2387                 cp = kdl->kdl_cache;
2388                 if (cp == NULL)
2389                         break;
2390                 if (kdl->kdl_alloc_fails)
2391                         ++warn;
2392                 if (header == 0) {
2393                         kmem_dumppr(&p, e,
2394                             "Cache Name,Allocs,Frees,Alloc Fails,"
2395                             "Nondump Frees,Unsafe Allocs/Frees\n");
2396                         header = 1;
2397                 }
2398                 kmem_dumppr(&p, e, "%s,%d,%d,%d,%d,%d\n",
2399                     cp->cache_name, kdl->kdl_allocs, kdl->kdl_frees,
2400                     kdl->kdl_alloc_fails, kdl->kdl_free_nondump,
2401                     kdl->kdl_unsafe);
2402         }

2404         /* return buffer size used */
2405         if (p < e)
2406                 bzero(p, e - p);
2407         return (p - buf);
2408 }

2410 /*
2411  * Allocate a constructed object from alternate dump memory.
2412  */
2413 void *
2414 kmem_cache_alloc_dump(kmem_cache_t *cp, int kmflag)
2415 {
2416         void *buf;
2417         void *curr;
2418         char *bufend;

2420         /* return a constructed object */
2421         if ((buf = cp->cache_dumpfreelist) != NULL) {
2422                 cp->cache_dumpfreelist = KMEM_DUMPCTL(cp, buf)->kdc_next;
2423                 KDI_LOG(cp, kdl_allocs);
2424                 return (buf);
2425         }

2427         /* create a new constructed object */
2428         curr = kmem_dump_curr;
2429         buf = (void *)P2ROUNDUP((uintptr_t)curr, cp->cache_align);
2430         bufend = (char *)KMEM_DUMPCTL(cp, buf) + sizeof (kmem_dumpctl_t);

2432         /* hat layer objects cannot cross a page boundary */
2433         if (cp->cache_align < PAGESIZE) {
```



```
2434                 char *page = (char *)P2ROUNDUP((uintptr_t)buf, PAGESIZE);
2435                 if (bufend > page) {
2436                         bufend += page - (char *)buf;
2437                         buf = (void *)page;
2438                 }
2439         }

2441         /* fall back to normal alloc if reserved area is used up */
2442         if (bufend > (char *)kmem_dump_end) {
2443                 kmem_dump_curr = kmem_dump_end;
2444                 KDI_LOG(cp, kdl_alloc_fails);
2445                 return (NULL);
2446         }

2448         /*
2449          * Must advance curr pointer before calling a constructor that
2450          * may also allocate memory.
2451          */
2452         kmem_dump_curr = bufend;

2454         /* run constructor */
2455         if (cp->cache_constructor != NULL &&
2456             cp->cache_constructor(buf, cp->cache_private, kmflag)
2457             != 0) {
2458 #ifdef DEBUG
2459                 printf("name='%s' cache=0x%p: kmem cache constructor failed\n",
2460                     cp->cache_name, (void *)cp);
2461 #endif
2462                 /* reset curr pointer iff no allocs were done */
2463                 if (kmem_dump_curr == bufend)
2464                         kmem_dump_curr = curr;

2466                 /* fall back to normal alloc if the constructor fails */
2467                 KDI_LOG(cp, kdl_alloc_fails);
2468                 return (NULL);
2469         }

2471         KDI_LOG(cp, kdl_allocs);
2472         return (buf);
2473 }

2475 /*
2476  * Free a constructed object in alternate dump memory.
2477  */
2478 int
2479 kmem_cache_free_dump(kmem_cache_t *cp, void *buf)
2480 {
2481         /* save constructed buffers for next time */
2482         if ((char *)buf >= (char *)kmem_dump_start &&
2483             (char *)buf < (char *)kmem_dump_end) {
2484                 KMEM_DUMPCTL(cp, buf)->kdc_next = cp->cache_dumpfreelist;
2485                 cp->cache_dumpfreelist = buf;
2486                 KDI_LOG(cp, kdl_frees);
2487                 return (0);
2488         }

2490         /* count all non-dump buf frees */
2491         KDI_LOG(cp, kdl_free_nondump);

2493         /* just drop buffers that were allocated before dump started */
2494         if (kmem_dump_curr < kmem_dump_end)
2495                 return (0);

2497         /* fall back to normal free if reserved area is used up */
2498         return (1);
2499 }
```

```
2501 /*
2502  * Allocate a constructed object from cache cp.
2503  */
2504 void *
2505 kmem_cache_alloc(kmem_cache_t *cp, int kmflag)
2506 {
2507         kmem_cpu_cache_t *ccp = KMEM_CPU_CACHE(cp);
2508         kmem_magazine_t *fmp;
2509         void *buf;

2511         mutex_enter(&ccp->cc_lock);
2512         for (;;) {
2513                 /*
2514                  * If there's an object available in the current CPU's
2515                  * loaded magazine, just take it and return.
2516                  */
2517                 if (ccp->cc_rounds > 0) {
2518                         buf = ccp->cc_loaded->mag_round[--ccp->cc_rounds];
2519                         ccp->cc_alloc++;
2520                         mutex_exit(&ccp->cc_lock);
2521                         if (ccp->cc_flags & (KMF_BUFTAG | KMF_DUMPUNSAFE)) {
2522                                 if (ccp->cc_flags & KMF_DUMPUNSAFE) {
2523                                         ASSERT(!(ccp->cc_flags &
2524                                             KMF_DUMPDIVERT));
2525                                         KDI_LOG(cp, kdl_unsafe);
2526                                 }
2527                                 if ((ccp->cc_flags & KMF_BUFTAG) &&
2528                                     kmem_cache_alloc_debug(cp, buf, kmflag, 0,
2529                                     caller()) != 0) {
2530                                         if (kmflag & KM_NOSLEEP)
2531                                                 return (NULL);
2532                                         mutex_enter(&ccp->cc_lock);
2533                                         continue;
2534                                 }
2535                         }
2536                         return (buf);
2537                 }

2539                 /*
2540                  * The loaded magazine is empty.  If the previously loaded
2541                  * magazine was full, exchange them and try again.
2542                  */
2543                 if (ccp->cc_prounds > 0) {
2544                         kmem_cpu_reload(ccp, ccp->cc_ploaded, ccp->cc_prounds);
2545                         continue;
2546                 }

2548                 /*
2549                  * Return an alternate buffer at dump time to preserve
2550                  * the heap.
2551                  */
2552                 if (ccp->cc_flags & (KMF_DUMPDIVERT | KMF_DUMPUNSAFE)) {
2553                         if (ccp->cc_flags & KMF_DUMPUNSAFE) {
2554                                 ASSERT(!(ccp->cc_flags & KMF_DUMPDIVERT));
2555                                 /* log it so that we can warn about it */
2556                                 KDI_LOG(cp, kdl_unsafe);
2557                         } else {
2558                                 if ((buf = kmem_cache_alloc_dump(cp, kmflag)) !=
2559                                     NULL) {
2560                                         mutex_exit(&ccp->cc_lock);
2561                                         return (buf);
2562                                 }
2563                                 break;          /* fall back to slab layer */
2564                         }
2565                 }
```



```
2567                 /*
2568                  * If the magazine layer is disabled, break out now.
2569                  */
2570                 if (ccp->cc_magsize == 0)
2571                         break;

2573                 /*
2574                  * Try to get a full magazine from the depot.
2575                  */
2576                 fmp = kmem_depot_alloc(cp, &cp->cache_full);
2577                 if (fmp != NULL) {
2578                         if (ccp->cc_ploaded != NULL)
2579                                 kmem_depot_free(cp, &cp->cache_empty,
2580                                     ccp->cc_ploaded);
2581                         kmem_cpu_reload(ccp, fmp, ccp->cc_magsize);
2582                         continue;
2583                 }

2585                 /*
2586                  * There are no full magazines in the depot,
2587                  * so fall through to the slab layer.
2588                  */
2589                 break;
2590         }
2591         mutex_exit(&ccp->cc_lock);

2593         /*
2594          * We couldn't allocate a constructed object from the magazine layer,
2595          * so get a raw buffer from the slab layer and apply its constructor.
2596          */
2597         buf = kmem_slab_alloc(cp, kmflag);

2599         if (buf == NULL)
2600                 return (NULL);

2602         if (cp->cache_flags & KMF_BUFTAG) {
2603                 /*
2604                  * Make kmem_cache_alloc_debug() apply the constructor for us.
2605                  */
2606                 int rc = kmem_cache_alloc_debug(cp, buf, kmflag, 1, caller());
2607                 if (rc != 0) {
2608                         if (kmflag & KM_NOSLEEP)
2609                                 return (NULL);
2610                         /*
2611                          * kmem_cache_alloc_debug() detected corruption
2612                          * but didn't panic (kmem_panic <= 0). We should not be
2613                          * here because the constructor failed (indicated by a
2614                          * return code of 1). Try again.
2615                          */
2616                         ASSERT(rc == -1);
2617                         return (kmem_cache_alloc(cp, kmflag));
2618                 }
2619                 return (buf);
2620         }

2622         if (cp->cache_constructor != NULL &&
2623             cp->cache_constructor(buf, cp->cache_private, kmflag) != 0) {
2624                 atomic_inc_64(&cp->cache_alloc_fail);
2625                 kmem_slab_free(cp, buf);
2626                 return (NULL);
2627         }

2629         return (buf);
2630 }
```

```
2632 /*
2633  * The freed argument tells whether or not kmem_cache_free_debug() has already
2634  * been called so that we can avoid the duplicate free error. For example, a
2635  * buffer on a magazine has already been freed by the client but is still
2636  * constructed.
2637  */
2638 static void
2639 kmem_slab_free_constructed(kmem_cache_t *cp, void *buf, boolean_t freed)
2640 {
2641         if (!freed && (cp->cache_flags & KMF_BUFTAG))
2642                 if (kmem_cache_free_debug(cp, buf, caller()) == -1)
2643                         return;
2644
2645         /*
2646          * Note that if KMF_DEADBEEF is in effect and KMF_LITE is not,
2647          * kmem_cache_free_debug() will have already applied the destructor.
2648          */
2649         if ((cp->cache_flags & (KMF_DEADBEEF | KMF_LITE)) != KMF_DEADBEEF &&
2650             cp->cache_destructor != NULL) {
2651                 if (cp->cache_flags & KMF_DEADBEEF) {   /* KMF_LITE implied */
2652                         kmem_buftag_t *btp = KMEM_BUFTAG(cp, buf);
2653                         *(uint64_t *)buf = btp->bt_redzone;
2654                         cp->cache_destructor(buf, cp->cache_private);
2655                         *(uint64_t *)buf = KMEM_FREE_PATTERN;
2656                 } else {
2657                         cp->cache_destructor(buf, cp->cache_private);
2658                 }
2659         }
2660
2661         kmem_slab_free(cp, buf);
2662 }
2663
2664 /*
2665  * Used when there's no room to free a buffer to the per-CPU cache.
2666  * Drops and re-acquires &ccp->cc_lock, and returns non-zero if the
2667  * caller should try freeing to the per-CPU cache again.
2668  * Note that we don't directly install the magazine in the cpu cache,
2669  * since its state may have changed wildly while the lock was dropped.
2670  */
2671 static int
2672 kmem_cpucache_magazine_alloc(kmem_cpu_cache_t *ccp, kmem_cache_t *cp)
2673 {
2674         kmem_magazine_t *emp;
2675         kmem_magtype_t *mtp;
2676
2677         ASSERT(MUTEX_HELD(&ccp->cc_lock));
2678         ASSERT(((uint_t)ccp->cc_rounds == ccp->cc_magsize ||
2679             ((uint_t)ccp->cc_rounds == -1)) &&
2680             ((uint_t)ccp->cc_prounds == ccp->cc_magsize ||
2681             ((uint_t)ccp->cc_prounds == -1)));
2682
2683         emp = kmem_depot_alloc(cp, &cp->cache_empty);
2684         if (emp != NULL) {
2685                 if (ccp->cc_ploaded != NULL)
2686                         kmem_depot_free(cp, &cp->cache_full,
2687                             ccp->cc_ploaded);
2688                 kmem_cpu_reload(ccp, emp, 0);
2689                 return (1);
2690         }
2691         /*
2692          * There are no empty magazines in the depot,
2693          * so try to allocate a new one.  We must drop all locks
2694          * across kmem_cache_alloc() because lower layers may
2695          * attempt to allocate from this cache.
2696          */
2697         mtp = cp->cache_magtype;
```



```
2698         mutex_exit(&ccp->cc_lock);
2699         emp = kmem_cache_alloc(mtp->mt_cache, KM_NOSLEEP);
2700         mutex_enter(&ccp->cc_lock);
2701
2702         if (emp != NULL) {
2703                 /*
2704                  * We successfully allocated an empty magazine.
2705                  * However, we had to drop ccp->cc_lock to do it,
2706                  * so the cache's magazine size may have changed.
2707                  * If so, free the magazine and try again.
2708                  */
2709                 if (ccp->cc_magsize != mtp->mt_magsize) {
2710                         mutex_exit(&ccp->cc_lock);
2711                         kmem_cache_free(mtp->mt_cache, emp);
2712                         mutex_enter(&ccp->cc_lock);
2713                         return (1);
2714                 }
2715
2716                 /*
2717                  * We got a magazine of the right size.  Add it to
2718                  * the depot and try the whole dance again.
2719                  */
2720                 kmem_depot_free(cp, &cp->cache_empty, emp);
2721                 return (1);
2722         }
2723
2724         /*
2725          * We couldn't allocate an empty magazine,
2726          * so fall through to the slab layer.
2727          */
2728         return (0);
2729 }
2730
2731 /*
2732  * Free a constructed object to cache cp.
2733  */
2734 void
2735 kmem_cache_free(kmem_cache_t *cp, void *buf)
2736 {
2737         kmem_cpu_cache_t *ccp = KMEM_CPU_CACHE(cp);
2738
2739         /*
2740          * The client must not free either of the buffers passed to the move
2741          * callback function.
2742          */
2743         ASSERT(cp->cache_defrag == NULL ||
2744             cp->cache_defrag->kmd_thread != curthread ||
2745             (buf != cp->cache_defrag->kmd_from_buf &&
2746             buf != cp->cache_defrag->kmd_to_buf));
2747
2748         if (ccp->cc_flags & (KMF_BUFTAG | KMF_DUMPDIVERT | KMF_DUMPUNSAFE)) {
2749                 if (ccp->cc_flags & KMF_DUMPUNSAFE) {
2750                         ASSERT(!(ccp->cc_flags & KMF_DUMPDIVERT));
2751                         /* log it so that we can warn about it */
2752                         KDI_LOG(cp, kdl_unsafe);
2753                 } else if (KMEM_DUMPCC(ccp) && !kmem_cache_free_dump(cp, buf)) {
2754                         return;
2755                 }
2756                 if (ccp->cc_flags & KMF_BUFTAG) {
2757                         if (kmem_cache_free_debug(cp, buf, caller()) == -1)
2758                                 return;
2759                 }
2760         }
2761
2762         mutex_enter(&ccp->cc_lock);
2763         /*
```

```
2764          * Any changes to this logic should be reflected in kmem_slab_prefill()
2765          */
2766         for (;;) {
2767                 /*
2768                  * If there's a slot available in the current CPU's
2769                  * loaded magazine, just put the object there and return.
2770                  */
2771                 if ((uint_t)ccp->cc_rounds < ccp->cc_magsize) {
2772                         ccp->cc_loaded->mag_round[ccp->cc_rounds++] = buf;
2773                         ccp->cc_free++;
2774                         mutex_exit(&ccp->cc_lock);
2775                         return;
2776                 }
2777
2778                 /*
2779                  * The loaded magazine is full.  If the previously loaded
2780                  * magazine was empty, exchange them and try again.
2781                  */
2782                 if (ccp->cc_prounds == 0) {
2783                         kmem_cpu_reload(ccp, ccp->cc_ploaded, ccp->cc_prounds);
2784                         continue;
2785                 }
2786
2787                 /*
2788                  * If the magazine layer is disabled, break out now.
2789                  */
2790                 if (ccp->cc_magsize == 0)
2791                         break;
2792
2793                 if (!kmem_cpucache_magazine_alloc(ccp, cp)) {
2794                         /*
2795                          * We couldn't free our constructed object to the
2796                          * magazine layer, so apply its destructor and free it
2797                          * to the slab layer.
2798                          */
2799                         break;
2800                 }
2801         }
2802         mutex_exit(&ccp->cc_lock);
2803         kmem_slab_free_constructed(cp, buf, B_TRUE);
2804 }
2805
2806 static void
2807 kmem_slab_prefill(kmem_cache_t *cp, kmem_slab_t *sp)
2808 {
2809         kmem_cpu_cache_t *ccp = KMEM_CPU_CACHE(cp);
2810         int cache_flags = cp->cache_flags;
2811
2812         kmem_bufctl_t *next, *head;
2813         size_t nbufs;
2814
2815         /*
2816          * Completely allocate the newly created slab and put the pre-allocated
2817          * buffers in magazines. Any of the buffers that cannot be put in
2818          * magazines must be returned to the slab.
2819          */
2820         ASSERT(MUTEX_HELD(&cp->cache_lock));
2821         ASSERT((cache_flags & (KMF_PREFILL|KMF_BUFTAG)) == KMF_PREFILL);
2822         ASSERT(cp->cache_constructor == NULL);
2823         ASSERT(sp->slab_cache == cp);
2824         ASSERT(sp->slab_refcnt == 1);
2825         ASSERT(sp->slab_head != NULL && sp->slab_chunks > sp->slab_refcnt);
2826         ASSERT(avl_find(&cp->cache_partial_slabs, sp, NULL) == NULL);
2827
2828         head = sp->slab_head;
2829         nbufs = (sp->slab_chunks - sp->slab_refcnt);
```



```
2830         sp->slab_head = NULL;
2831         sp->slab_refcnt += nbufs;
2832         cp->cache_bufslab -= nbufs;
2833         cp->cache_slab_alloc += nbufs;
2834         list_insert_head(&cp->cache_complete_slabs, sp);
2835         cp->cache_complete_slab_count++;
2836         mutex_exit(&cp->cache_lock);
2837         mutex_enter(&ccp->cc_lock);
2838
2839         while (head != NULL) {
2840                 void *buf = KMEM_BUF(cp, head);
2841                 /*
2842                  * If there's a slot available in the current CPU's
2843                  * loaded magazine, just put the object there and
2844                  * continue.
2845                  */
2846                 if ((uint_t)ccp->cc_rounds < ccp->cc_magsize) {
2847                         ccp->cc_loaded->mag_round[ccp->cc_rounds++] =
2848                             buf;
2849                         ccp->cc_free++;
2850                         nbufs--;
2851                         head = head->bc_next;
2852                         continue;
2853                 }
2854
2855                 /*
2856                  * The loaded magazine is full.  If the previously
2857                  * loaded magazine was empty, exchange them and try
2858                  * again.
2859                  */
2860                 if (ccp->cc_prounds == 0) {
2861                         kmem_cpu_reload(ccp, ccp->cc_ploaded,
2862                             ccp->cc_prounds);
2863                         continue;
2864                 }
2865
2866                 /*
2867                  * If the magazine layer is disabled, break out now.
2868                  */
2869
2870                 if (ccp->cc_magsize == 0) {
2871                         break;
2872                 }
2873
2874                 if (!kmem_cpucache_magazine_alloc(ccp, cp))
2875                         break;
2876         }
2877         mutex_exit(&ccp->cc_lock);
2878         if (nbufs != 0) {
2879                 ASSERT(head != NULL);
2880
2881                 /*
2882                  * If there was a failure, return remaining objects to
2883                  * the slab
2884                  */
2885                 while (head != NULL) {
2886                         ASSERT(nbufs != 0);
2887                         next = head->bc_next;
2888                         head->bc_next = NULL;
2889                         kmem_slab_free(cp, KMEM_BUF(cp, head));
2890                         head = next;
2891                         nbufs--;
2892                 }
2893         }
2894         ASSERT(head == NULL);
2895         ASSERT(nbufs == 0);
```

```
2896         mutex_enter(&cp->cache_lock);
2897 }

2899 void *
2900 kmem_zalloc(size_t size, int kmflag)
2901 {
2902         size_t index;
2903         void *buf;

2905         if ((index = ((size - 1) >> KMEM_ALIGN_SHIFT)) < KMEM_ALLOC_TABLE_MAX) {
2906                 kmem_cache_t *cp = kmem_alloc_table[index];
2907                 buf = kmem_cache_alloc(cp, kmflag);
2908                 if (buf != NULL) {
2909                         if ((cp->cache_flags & KMF_BUFTAG) && !KMEM_DUMP(cp)) {
2910                                 kmem_buftag_t *btp = KMEM_BUFTAG(cp, buf);
2911                                 ((uint8_t *)buf)[size] = KMEM_REDZONE_BYTE;
2912                                 ((uint32_t *)btp)[1] = KMEM_SIZE_ENCODE(size);

2914                                 if (cp->cache_flags & KMF_LITE) {
2915                                         KMEM_BUFTAG_LITE_ENTER(btp,
2916                                             kmem_lite_count, caller());
2917                                 }
2918                         }
2919                         bzero(buf, size);
2920                 }
2921         } else {
2922                 buf = kmem_alloc(size, kmflag);
2923                 if (buf != NULL)
2924                         bzero(buf, size);
2925         }
2926         return (buf);
2927 }

2929 void *
2930 kmem_alloc(size_t size, int kmflag)
2931 {
2932         size_t index;
2933         kmem_cache_t *cp;
2934         void *buf;

2936         if ((index = ((size - 1) >> KMEM_ALIGN_SHIFT)) < KMEM_ALLOC_TABLE_MAX) {
2937                 cp = kmem_alloc_table[index];
2938                 /* fall through to kmem_cache_alloc() */

2940         } else if ((index = ((size - 1) >> KMEM_BIG_SHIFT)) <
2941             kmem_big_alloc_table_max) {
2942                 cp = kmem_big_alloc_table[index];
2943                 /* fall through to kmem_cache_alloc() */

2945         } else {
2946                 if (size == 0)
2947                         return (NULL);

2949                 buf = vmem_alloc(kmem_oversize_arena, size,
2950                     kmflag & KM_VMFLAGS);
2951                 if (buf == NULL)
2952                         kmem_log_event(kmem_failure_log, NULL, NULL,
2953                             (void *)size);
2954                 else if (KMEM_DUMP(kmem_slab_cache)) {
2955                         /* stats for dump intercept */
2956                         kmem_dump_oversize_allocs++;
2957                         if (size > kmem_dump_oversize_max)
2958                                 kmem_dump_oversize_max = size;
2959                 }
2960                 return (buf);
2961         }
```



```
2963         buf = kmem_cache_alloc(cp, kmflag);
2964         if ((cp->cache_flags & KMF_BUFTAG) && !KMEM_DUMP(cp) && buf != NULL) {
2965                 kmem_buftag_t *btp = KMEM_BUFTAG(cp, buf);
2966                 ((uint8_t *)buf)[size] = KMEM_REDZONE_BYTE;
2967                 ((uint32_t *)btp)[1] = KMEM_SIZE_ENCODE(size);

2969                 if (cp->cache_flags & KMF_LITE) {
2970                         KMEM_BUFTAG_LITE_ENTER(btp, kmem_lite_count, caller());
2971                 }
2972         }
2973         return (buf);
2974 }

2976 void
2977 kmem_free(void *buf, size_t size)
2978 {
2979         size_t index;
2980         kmem_cache_t *cp;

2982         if ((index = (size - 1) >> KMEM_ALIGN_SHIFT) < KMEM_ALLOC_TABLE_MAX) {
2983                 cp = kmem_alloc_table[index];
2984                 /* fall through to kmem_cache_free() */

2986         } else if ((index = ((size - 1) >> KMEM_BIG_SHIFT)) <
2987             kmem_big_alloc_table_max) {
2988                 cp = kmem_big_alloc_table[index];
2989                 /* fall through to kmem_cache_free() */

2991         } else {
2992                 EQUIV(buf == NULL, size == 0);
2993                 if (buf == NULL && size == 0)
2994                         return;
2995                 vmem_free(kmem_oversize_arena, buf, size);
2996                 return;
2997         }

2999         if ((cp->cache_flags & KMF_BUFTAG) && !KMEM_DUMP(cp)) {
3000                 kmem_buftag_t *btp = KMEM_BUFTAG(cp, buf);
3001                 uint32_t *ip = (uint32_t *)btp;
3002                 if (ip[1] != KMEM_SIZE_ENCODE(size)) {
3003                         if (*(uint64_t *)buf == KMEM_FREE_PATTERN) {
3004                                 kmem_error(KMERR_DUPFREE, cp, buf);
3005                                 return;
3006                         }
3007                         if (KMEM_SIZE_VALID(ip[1])) {
3008                                 ip[0] = KMEM_SIZE_ENCODE(size);
3009                                 kmem_error(KMERR_BADSIZE, cp, buf);
3010                         } else {
3011                                 kmem_error(KMERR_REDZONE, cp, buf);
3012                         }
3013                         return;
3014                 }
3015                 if (((uint8_t *)buf)[size] != KMEM_REDZONE_BYTE) {
3016                         kmem_error(KMERR_REDZONE, cp, buf);
3017                         return;
3018                 }
3019                 btp->bt_redzone = KMEM_REDZONE_PATTERN;
3020                 if (cp->cache_flags & KMF_LITE) {
3021                         KMEM_BUFTAG_LITE_ENTER(btp, kmem_lite_count,
3022                             caller());
3023                 }
3024         }
3025         kmem_cache_free(cp, buf);
3026 }
```

```
3028 void *
3029 kmem_firewall_va_alloc(vmem_t *vmp, size_t size, int vmflag)
3030 {
3031         size_t realsize = size + vmp->vm_quantum;
3032         void *addr;
3033 
3034         /*
3035          * Annoying edge case: if 'size' is just shy of ULONG_MAX, adding
3036          * vm_quantum will cause integer wraparound.  Check for this, and
3037          * blow off the firewall page in this case.  Note that such a
3038          * giant allocation (the entire kernel address space) can never
3039          * be satisfied, so it will either fail immediately (VM_NOSLEEP)
3040          * or sleep forever (VM_SLEEP).  Thus, there is no need for a
3041          * corresponding check in kmem_firewall_va_free().
3042          */
3043         if (realsize < size)
3044                 realsize = size;
3045 
3046         /*
3047          * While boot still owns resource management, make sure that this
3048          * redzone virtual address allocation is properly accounted for in
3049          * OBPs "virtual-memory" "available" lists because we're
3050          * effectively claiming them for a red zone.  If we don't do this,
3051          * the available lists become too fragmented and too large for the
3052          * current boot/kernel memory list interface.
3053          */
3054         addr = vmem_alloc(vmp, realsize, vmflag | VM_NEXTFIT);
3055 
3056         if (addr != NULL && kvseg.s_base == NULL && realsize != size)
3057                 (void) boot_virt_alloc((char *)addr + size, vmp->vm_quantum);
3058 
3059         return (addr);
3060 }
3061 
3062 void
3063 kmem_firewall_va_free(vmem_t *vmp, void *addr, size_t size)
3064 {
3065         ASSERT((kvseg.s_base == NULL ?
3066             va_to_pfn((char *)addr + size) :
3067             hat_getpfnum(kas.a_hat, (caddr_t)addr + size)) == PFN_INVALID);
3068 
3069         vmem_free(vmp, addr, size + vmp->vm_quantum);
3070 }
3071 
3072 /*
3073  * Try to allocate at least 'size' bytes of memory without sleeping or
3074  * panicking. Return actual allocated size in 'asize'. If allocation failed,
3075  * try final allocation with sleep or panic allowed.
3076  */
3077 void *
3078 kmem_alloc_tryhard(size_t size, size_t *asize, int kmflag)
3079 {
3080         void *p;
3081 
3082         *asize = P2ROUNDUP(size, KMEM_ALIGN);
3083         do {
3084                 p = kmem_alloc(*asize, (kmflag | KM_NOSLEEP) & ~KM_PANIC);
3085                 if (p != NULL)
3086                         return (p);
3087                 *asize += KMEM_ALIGN;
3088         } while (*asize <= PAGESIZE);
3089 
3090         *asize = P2ROUNDUP(size, KMEM_ALIGN);
3091         return (kmem_alloc(*asize, kmflag));
3092 }
```



```
3094 /*
3095  * Reclaim all unused memory from a cache.
3096  */
3097 static void
3098 kmem_cache_reap(kmem_cache_t *cp)
3099 {
3100         ASSERT(taskq_member(kmem_taskq, curthread));
3101         cp->cache_reap++;
3102 
3103         /*
3104          * Ask the cache's owner to free some memory if possible.
3105          * The idea is to handle things like the inode cache, which
3106          * typically sits on a bunch of memory that it doesn't truly
3107          * *need*.  Reclaim policy is entirely up to the owner; this
3108          * callback is just an advisory plea for help.
3109          */
3110         if (cp->cache_reclaim != NULL) {
3111                 long delta;
3112 
3113                 /*
3114                  * Reclaimed memory should be reapable (not included in the
3115                  * depot's working set).
3116                  */
3117                 delta = cp->cache_full.ml_total;
3118                 cp->cache_reclaim(cp->cache_private);
3119                 delta = cp->cache_full.ml_total - delta;
3120                 if (delta > 0) {
3121                         mutex_enter(&cp->cache_depot_lock);
3122                         cp->cache_full.ml_reaplimit += delta;
3123                         cp->cache_full.ml_min += delta;
3124                         mutex_exit(&cp->cache_depot_lock);
3125                 }
3126         }
3127 
3128         kmem_depot_ws_reap(cp);
3129 
3130         if (cp->cache_defrag != NULL && !kmem_move_noreap) {
3131                 kmem_cache_defrag(cp);
3132         }
3133 }
3134 
3135 static void
3136 kmem_reap_timeout(void *flag_arg)
3137 {
3138         uint32_t *flag = (uint32_t *)flag_arg;
3139 
3140         ASSERT(flag == &kmem_reaping || flag == &kmem_reaping_idspace);
3141         *flag = 0;
3142 }
3143 
3144 static void
3145 kmem_reap_done(void *flag)
3146 {
3147         if (!callout_init_done) {
3148                 /* can't schedule a timeout at this point */
3149                 kmem_reap_timeout(flag);
3150         } else {
3151                 (void) timeout(kmem_reap_timeout, flag, kmem_reap_interval);
3152         }
3153 }
3154 
3155 static void
3156 kmem_reap_start(void *flag)
3157 {
3158         ASSERT(flag == &kmem_reaping || flag == &kmem_reaping_idspace);
```

```
3160         if (flag == &kmem_reaping) {
3161                 kmem_cache_applyall(kmem_cache_reap, kmem_taskq, TQ_NOSLEEP);
3162                 /*
3163                  * if we have segkp under heap, reap segkp cache.
3164                  */
3165                 if (segkp_fromheap)
3166                         segkp_cache_free();
3167         }
3168         else
3169                 kmem_cache_applyall_id(kmem_cache_reap, kmem_taskq, TQ_NOSLEEP);

3171         /*
3172          * We use taskq_dispatch() to schedule a timeout to clear
3173          * the flag so that kmem_reap() becomes self-throttling:
3174          * we won't reap again until the current reap completes *and*
3175          * at least kmem_reap_interval ticks have elapsed.
3176          */
3177         if (!taskq_dispatch(kmem_taskq, kmem_reap_done, flag, TQ_NOSLEEP))
3178                 kmem_reap_done(flag);
3179 }

3181 static void
3182 kmem_reap_common(void *flag_arg)
3183 {
3184         uint32_t *flag = (uint32_t *)flag_arg;

3186         if (MUTEX_HELD(&kmem_cache_lock) || kmem_taskq == NULL ||
3187             atomic_cas_32(flag, 0, 1) != 0)
3188                 return;

3190         /*
3191          * It may not be kosher to do memory allocation when a reap is called
3192          * (for example, if vmem_populate() is in the call chain).  So we
3193          * start the reap going with a TQ_NOALLOC dispatch.  If the dispatch
3194          * fails, we reset the flag, and the next reap will try again.
3195          */
3196         if (!taskq_dispatch(kmem_taskq, kmem_reap_start, flag, TQ_NOALLOC))
3197                 *flag = 0;
3198 }

3200 /*
3201  * Reclaim all unused memory from all caches.  Called from the VM system
3202  * when memory gets tight.
3203  */
3204 void
3205 kmem_reap(void)
3206 {
3207         kmem_reap_common(&kmem_reaping);
3208 }

3210 /*
3211  * Reclaim all unused memory from identifier arenas, called when a vmem
3212  * arena not back by memory is exhausted.  Since reaping memory-backed caches
3213  * cannot help with identifier exhaustion, we avoid both a large amount of
3214  * work and unwanted side-effects from reclaim callbacks.
3215  */
3216 void
3217 kmem_reap_idspace(void)
3218 {
3219         kmem_reap_common(&kmem_reaping_idspace);
3220 }

3222 /*
3223  * Purge all magazines from a cache and set its magazine limit to zero.
3224  * All calls are serialized by the kmem_taskq lock, except for the final
3225  * call from kmem_cache_destroy().
```



```
3226  */
3227 static void
3228 kmem_cache_magazine_purge(kmem_cache_t *cp)
3229 {
3230         kmem_cpu_cache_t *ccp;
3231         kmem_magazine_t *mp, *pmp;
3232         int rounds, prounds, cpu_seqid;

3234         ASSERT(!list_link_active(&cp->cache_link) ||
3235             taskq_member(kmem_taskq, curthread));
3236         ASSERT(MUTEX_NOT_HELD(&cp->cache_lock));

3238         for (cpu_seqid = 0; cpu_seqid < max_ncpus; cpu_seqid++) {
3239                 ccp = &cp->cache_cpu[cpu_seqid];

3241                 mutex_enter(&ccp->cc_lock);
3242                 mp = ccp->cc_loaded;
3243                 pmp = ccp->cc_ploaded;
3244                 rounds = ccp->cc_rounds;
3245                 prounds = ccp->cc_prounds;
3246                 ccp->cc_loaded = NULL;
3247                 ccp->cc_ploaded = NULL;
3248                 ccp->cc_rounds = -1;
3249                 ccp->cc_prounds = -1;
3250                 ccp->cc_magsize = 0;
3251                 mutex_exit(&ccp->cc_lock);

3253                 if (mp)
3254                         kmem_magazine_destroy(cp, mp, rounds);
3255                 if (pmp)
3256                         kmem_magazine_destroy(cp, pmp, prounds);
3257         }

3259         kmem_depot_ws_zero(cp);
  23         /*
  24          * Updating the working set statistics twice in a row has the
  25          * effect of setting the working set size to zero, so everything
  26          * is eligible for reaping.
  27          */
  28         kmem_depot_ws_update(cp);
  29         kmem_depot_ws_update(cp);

3260         kmem_depot_ws_reap(cp);
3261 }
______unchanged_portion_omitted_

3283 /*
3284  * Reap (almost) everything right now.
  55  * Reap (almost) everything right now.  See kmem_cache_magazine_purge()
  56  * for explanation of the back-to-back kmem_depot_ws_update() calls.
3285  */
3286 void
3287 kmem_cache_reap_now(kmem_cache_t *cp)
3288 {
3289         ASSERT(list_link_active(&cp->cache_link));

3291         kmem_depot_ws_zero(cp);
  63         kmem_depot_ws_update(cp);
  64         kmem_depot_ws_update(cp);

3293         (void) taskq_dispatch(kmem_taskq,
3294             (task_func_t *)kmem_depot_ws_reap, cp, TQ_SLEEP);
3295         taskq_wait(kmem_taskq);
3296 }
______unchanged_portion_omitted_
```