```
**********************************************************
   70955 Sat May 30 10:23:51 2015
new/usr/src/uts/common/os/taskq.c
5881 corrected maxall vs. maxalloc in comments
**********************************************************
```

```
/*
 * 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 2010 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 */

/*
 * Copyright 2015 Nexenta Systems, Inc.  All rights reserved.
 */

/*
 * Kernel task queues: general-purpose asynchronous task scheduling.
 *
 * A common problem in kernel programming is the need to schedule tasks
 * to be performed later, by another thread. There are several reasons
 * you may want or need to do this:
 *
 * (1) The task isn't time-critical, but your current code path is.
 *
 * (2) The task may require grabbing locks that you already hold.
 *
 * (3) The task may need to block (e.g. to wait for memory), but you
 *     cannot block in your current context.
 *
 * (4) Your code path can't complete because of some condition, but you can't
 *     sleep or fail, so you queue the task for later execution when condition
 *     disappears.
 *
 * (5) You just want a simple way to launch multiple tasks in parallel.
 *
 * Task queues provide such a facility. In its simplest form (used when
 * performance is not a critical consideration) a task queue consists of a
 * single list of tasks, together with one or more threads to service the
 * list. There are some cases when this simple queue is not sufficient:
 *
 * (1) The task queues are very hot and there is a need to avoid data and lock
 *      contention over global resources.
 *
 * (2) Some tasks may depend on other tasks to complete, so they can't be put in
 *      the same list managed by the same thread.
 *
 * (3) Some tasks may block for a long time, and this should not block other
```

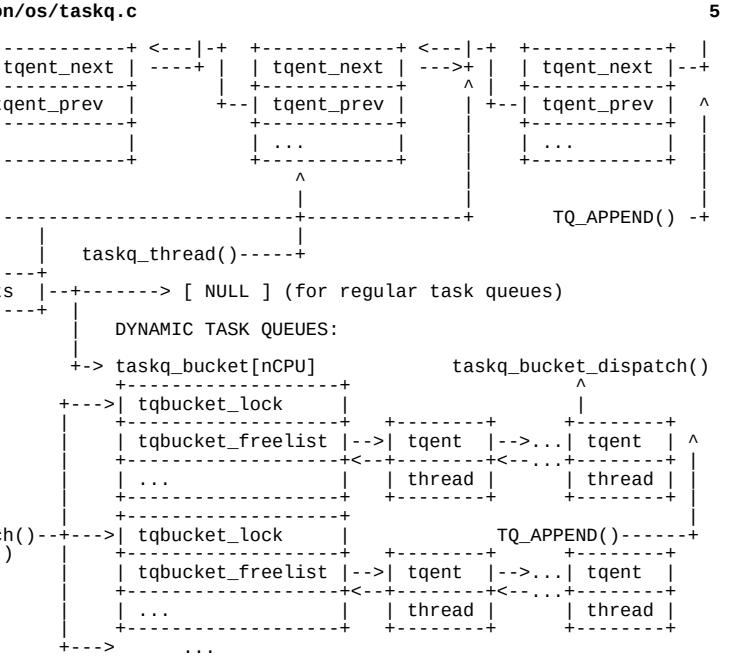


```
 *      tasks in the queue.
 *
 * To provide useful service in such cases we define a "dynamic task queue"
 * which has an individual thread for each of the tasks. These threads are
 * dynamically created as they are needed and destroyed when they are not in
 * use. The API for managing task pools is the same as for managing task queues
 * with the exception of a taskq creation flag TASKQ_DYNAMIC which tells that
 * dynamic task pool behavior is desired.
 *
 * Dynamic task queues may also place tasks in the normal queue (called "backing
 * queue") when task pool runs out of resources. Users of task queues may
 * disallow such queued scheduling by specifying TQ_NOQUEUE in the dispatch
 * flags.
 *
 * The backing task queue is also used for scheduling internal tasks needed for
 * dynamic task queue maintenance.
 *
 * INTERFACES ==================================================================
 *
 * taskq_t *taskq_create(name, nthreads, pri, minalloc, maxalloc, flags);
 * taskq_t *taskq_create(name, nthreads, pri, minalloc, maxall, flags);
 *
 *      Create a taskq with specified properties.
 *      Possible 'flags':
 *
 *        TASKQ_DYNAMIC: Create task pool for task management. If this flag is
 *              specified, 'nthreads' specifies the maximum number of threads in
 *              the task queue. Task execution order for dynamic task queues is
 *              not predictable.
 *
 *              If this flag is not specified (default case) a
 *              single-list task queue is created with 'nthreads' threads
 *              servicing it. Entries in this queue are managed by
 *              taskq_ent_alloc() and taskq_ent_free() which try to keep the
 *              task population between 'minalloc' and 'maxalloc', but the
 *              latter limit is only advisory for TQ_SLEEP dispatches and the
 *              former limit is only advisory for TQ_NOALLOC dispatches. If
 *              TASKQ_PREPOPULATE is set in 'flags', the taskq will be
 *              prepopulated with 'minalloc' task structures.
 *
 *              Since non-DYNAMIC taskqs are queues, tasks are guaranteed to be
 *              executed in the order they are scheduled if nthreads == 1.
 *              If nthreads > 1, task execution order is not predictable.
 *
 *        TASKQ_PREPOPULATE: Prepopulate task queue with threads.
 *              Also prepopulate the task queue with 'minalloc' task structures.
 *
 *        TASKQ_THREADS_CPU_PCT: This flag specifies that 'nthreads' should be
 *              interpreted as a percentage of the # of online CPUs on the
 *              system.  The taskq subsystem will automatically adjust the
 *              number of threads in the taskq in response to CPU online
 *              and offline events, to keep the ratio.  nthreads must be in
 *              the range [0,100].
 *
 *              The calculation used is:
 *
 *                      MAX((ncpus_online * percentage)/100, 1)
 *
 *              This flag is not supported for DYNAMIC task queues.
 *              This flag is not compatible with TASKQ_CPR_SAFE.
 *
 *        TASKQ_CPR_SAFE: This flag specifies that users of the task queue will
 *              use their own protocol for handling CPR issues. This flag is not
 *              supported for DYNAMIC task queues.  This flag is not compatible
 *              with TASKQ_THREADS_CPU_PCT.
 *
```

```
 127  *      The 'pri' field specifies the default priority for the threads that
 128  *      service all scheduled tasks.
 129  *
 130  * taskq_t *taskq_create_instance(name, instance, nthreads, pri, minalloc,
 131  *    maxalloc, flags);
 131  *    maxall, flags);
 132  *
 133  *      Like taskq_create(), but takes an instance number (or -1 to indicate
 134  *      no instance).
 135  *
 136  * taskq_t *taskq_create_proc(name, nthreads, pri, minalloc, maxalloc, proc,
 136  * taskq_t *taskq_create_proc(name, nthreads, pri, minalloc, maxall, proc,
 137  *    flags);
 138  *
 139  *      Like taskq_create(), but creates the taskq threads in the specified
 140  *      system process.  If proc != &p0, this must be called from a thread
 141  *      in that process.
 142  *
 143  * taskq_t *taskq_create_sysdc(name, nthreads, minalloc, maxalloc, proc,
 143  * taskq_t *taskq_create_sysdc(name, nthreads, minalloc, maxall, proc,
 144  *    dc, flags);
 145  *
 146  *      Like taskq_create_proc(), but the taskq threads will use the
 147  *      System Duty Cycle (SDC) scheduling class with a duty cycle of dc.
 148  *
 149  * void taskq_destroy(tap):
 150  *
 151  *      Waits for any scheduled tasks to complete, then destroys the taskq.
 152  *      Caller should guarantee that no new tasks are scheduled in the closing
 153  *      taskq.
 154  *
 155  * taskqid_t taskq_dispatch(tq, func, arg, flags):
 156  *
 157  *      Dispatches the task "func(arg)" to taskq. The 'flags' indicates whether
 158  *      the caller is willing to block for memory.  The function returns an
 159  *      opaque value which is zero iff dispatch fails.  If flags is TQ_NOSLEEP
 160  *      or TQ_NOALLOC and the task can't be dispatched, taskq_dispatch() fails
 161  *      and returns (taskqid_t)0.
 162  *
 163  *      ASSUMES: func != NULL.
 164  *
 165  *      Possible flags:
 166  *        TQ_NOSLEEP: Do not wait for resources; may fail.
 167  *
 168  *        TQ_NOALLOC: Do not allocate memory; may fail.  May only be used with
 169  *              non-dynamic task queues.
 170  *
 171  *        TQ_NOQUEUE: Do not enqueue a task if it can't dispatch it due to
 172  *              lack of available resources and fail. If this flag is not
 173  *              set, and the task pool is exhausted, the task may be scheduled
 174  *              in the backing queue. This flag may ONLY be used with dynamic
 175  *              task queues.
 176  *
 177  *              NOTE: This flag should always be used when a task queue is used
 178  *              for tasks that may depend on each other for completion.
 179  *              Enqueueing dependent tasks may create deadlocks.
 180  *
 181  *        TQ_SLEEP:   May block waiting for resources. May still fail for
 182  *              dynamic task queues if TQ_NOQUEUE is also specified, otherwise
 183  *              always succeed.
 184  *
 185  *        TQ_FRONT:   Puts the new task at the front of the queue.  Be careful.
 186  *
 187  *      NOTE: Dynamic task queues are much more likely to fail in
 188  *              taskq_dispatch() (especially if TQ_NOQUEUE was specified), so it
 189  *              is important to have backup strategies handling such failures.
```



```
 190  *
 191  * void taskq_dispatch_ent(tq, func, arg, flags, tqent)
 192  *
 193  *      This is a light-weight form of taskq_dispatch(), that uses a
 194  *      preallocated taskq_ent_t structure for scheduling.  As a
 195  *      result, it does not perform allocations and cannot ever fail.
 196  *      Note especially that it cannot be used with TASKQ_DYNAMIC
 197  *      taskqs.  The memory for the tqent must not be modified or used
 198  *      until the function (func) is called.  (However, func itself
 199  *      may safely modify or free this memory, once it is called.)
 200  *      Note that the taskq framework will NOT free this memory.
 201  *
 202  * void taskq_wait(tq):
 203  *
 204  *      Waits for all previously scheduled tasks to complete.
 205  *
 206  *      NOTE: It does not stop any new task dispatches.
 207  *            Do NOT call taskq_wait() from a task: it will cause deadlock.
 208  *
 209  * void taskq_suspend(tq)
 210  *
 211  *      Suspend all task execution. Tasks already scheduled for a dynamic task
 212  *      queue will still be executed, but all new scheduled tasks will be
 213  *      suspended until taskq_resume() is called.
 214  *
 215  * int  taskq_suspended(tq)
 216  *
 217  *      Returns 1 if taskq is suspended and 0 otherwise. It is intended to
 218  *      ASSERT that the task queue is suspended.
 219  *
 220  * void taskq_resume(tq)
 221  *
 222  *      Resume task queue execution.
 223  *
 224  * int  taskq_member(tq, thread)
 225  *
 226  *      Returns 1 if 'thread' belongs to taskq 'tq' and 0 otherwise. The
 227  *      intended use is to ASSERT that a given function is called in taskq
 228  *      context only.
 229  *
 230  * system_taskq
 231  *
 232  *      Global system-wide dynamic task queue for common uses. It may be used by
 233  *      any subsystem that needs to schedule tasks and does not need to manage
 234  *      its own task queues. It is initialized quite early during system boot.
 235  *
 236  * IMPLEMENTATION ================================================================
 237  *
 238  * This is schematic representation of the task queue structures.
 239  *
 240  *   taskq:
 241  *   +-------------+
 242  *   | tq_lock     | +---< taskq_ent_free()
 243  *   +-------------+ |
 244  *   |...          | | tqent:                tqent:
 245  *   +-------------+ | +------------+        +------------+
 246  *   | tq_freelist |-->| tqent_next |--> ... ->| tqent_next |
 247  *   +-------------+   +------------+        +------------+
 248  *   |...          |   | ...        |        | ...        |
 249  *   +-------------+   +------------+        +------------+
 250  *   | tq_task     |    |
 251  *   |             |    +------------->taskq_ent_alloc()
 252  *  +--------------------------------------------------------------------------+
 253  *  | |                       |           tqent                  tqent        |
 254  *  | +---------------------+     +--> +------------+     +--> +------------+  |
 255  *  | | ...                 |     |    | func, arg  |     |    | func, arg  |  |
```



```
 * Task queues use tq_task field to link new entry in the queue. The queue is a
 * circular doubly-linked list. Entries are put in the end of the list with
 * TQ_APPEND() and processed from the front of the list by taskq_thread() in
 * FIFO order. Task queue entries are cached in the free list managed by
 * taskq_ent_alloc() and taskq_ent_free() functions.
 *
 *      All threads used by task queues mark t_taskq field of the thread to
 *      point to the task queue.
 *
 * Taskq Thread Management -----------------------------------------------------
 *
 * Taskq's non-dynamic threads are managed with several variables and flags:
 *
 *      * tq_nthreads   - The number of threads in taskq_thread() for the
 *                        taskq.
 *
 *      * tq_active     - The number of threads not waiting on a CV in
 *                        taskq_thread(); includes newly created threads
 *                        not yet counted in tq_nthreads.
 *
 *      * tq_nthreads_target
 *                      - The number of threads desired for the taskq.
 *
 *      * tq_flags & TASKQ_CHANGING
 *                      - Indicates that tq_nthreads != tq_nthreads_target.
 *
 *      * tq_flags & TASKQ_THREAD_CREATED
 *                      - Indicates that a thread is being created in the taskq.
 *
 * During creation, tq_nthreads and tq_active are set to 0, and
 * tq_nthreads_target is set to the number of threads desired.  The
```



```
 * TASKQ_CHANGING flag is set, and taskq_thread_create() is called to
 * create the first thread. taskq_thread_create() increments tq_active,
 * sets TASKQ_THREAD_CREATED, and creates the new thread.
 *
 * Each thread starts in taskq_thread(), clears the TASKQ_THREAD_CREATED
 * flag, and increments tq_nthreads.  It stores the new value of
 * tq_nthreads as its "thread_id", and stores its thread pointer in the
 * tq_threadlist at the (thread_id - 1).  We keep the thread_id space
 * densely packed by requiring that only the largest thread_id can exit during
 * normal adjustment.   The exception is during the destruction of the
 * taskq; once tq_nthreads_target is set to zero, no new threads will be created
 * for the taskq queue, so every thread can exit without any ordering being
 * necessary.
 *
 * Threads will only process work if their thread id is <= tq_nthreads_target.
 *
 * When TASKQ_CHANGING is set, threads will check the current thread target
 * whenever they wake up, and do whatever they can to apply its effects.
 *
 * TASKQ_THREAD_CPU_PCT -------------------------------------------------------
 *
 * When a taskq is created with TASKQ_THREAD_CPU_PCT, we store their requested
 * percentage in tq_threads_ncpus_pct, start them off with the correct thread
 * target, and add them to the taskq_cpupct_list for later adjustment.
 *
 * We register taskq_cpu_setup() to be called whenever a CPU changes state.  It
 * walks the list of TASKQ_THREAD_CPU_PCT taskqs, adjusts their nthread_target
 * if need be, and wakes up all of the threads to process the change.
 *
 * Dynamic Task Queues Implementation -----------------------------------------
 *
 * For a dynamic task queues there is a 1-to-1 mapping between a thread and
 * taskq_ent_structure. Each entry is serviced by its own thread and each thread
 * is controlled by a single entry.
 *
 * Entries are distributed over a set of buckets. To avoid using modulo
 * arithmetics the number of buckets is 2^n and is determined as the nearest
 * power of two roundown of the number of CPUs in the system. Tunable
 * variable 'taskq_maxbuckets' limits the maximum number of buckets. Each entry
 * is attached to a bucket for its lifetime and can't migrate to other buckets.
 *
 * Entries that have scheduled tasks are not placed in any list. The dispatch
 * function sets their "func" and "arg" fields and signals the corresponding
 * thread to execute the task. Once the thread executes the task it clears the
 * "func" field and places an entry on the bucket cache of free entries pointed
 * by "tqbucket_freelist" field. ALL entries on the free list should have "func"
 * field equal to NULL. The free list is a circular doubly-linked list identical
 * in structure to the tq_task list above, but entries are taken from it in LIFO
 * order - the last freed entry is the first to be allocated. The
 * taskq_bucket_dispatch() function gets the most recently used entry from the
 * free list, sets its "func" and "arg" fields and signals a worker thread.
 *
 * After executing each task a per-entry thread taskq_d_thread() places its
 * entry on the bucket free list and goes to a timed sleep. If it wakes up
 * without getting new task it removes the entry from the free list and destroys
 * itself. The thread sleep time is controlled by a tunable variable
 * 'taskq_thread_timeout'.
 *
 * There are various statistics kept in the bucket which allows for later
 * analysis of taskq usage patterns. Also, a global copy of taskq creation and
 * death statistics is kept in the global taskq data structure. Since thread
 * creation and death happen rarely, updating such global data does not present
 * a performance problem.
 *
 * NOTE: Threads are not bound to any CPU and there is absolutely no association
 *       between the bucket and actual thread CPU, so buckets are used only to
```

```
 388  *       split resources and reduce resource contention. Having threads attached
 389  *       to the CPU denoted by a bucket may reduce number of times the job
 390  *       switches between CPUs.
 391  *
 392  *       Current algorithm creates a thread whenever a bucket has no free
 393  *       entries. It would be nice to know how many threads are in the running
 394  *       state and don't create threads if all CPUs are busy with existing
 395  *       tasks, but it is unclear how such strategy can be implemented.
 396  *
 397  *       Currently buckets are created statically as an array attached to task
 398  *       queue. On some system with nCPUs < max_ncpus it may waste system
 399  *       memory. One solution may be allocation of buckets when they are first
 400  *       touched, but it is not clear how useful it is.
 401  *
 402  * SUSPEND/RESUME implementation -----------------------------------------------
 403  *
 404  *      Before executing a task taskq_thread() (executing non-dynamic task
 405  *      queues) obtains taskq's thread lock as a reader. The taskq_suspend()
 406  *      function gets the same lock as a writer blocking all non-dynamic task
 407  *      execution. The taskq_resume() function releases the lock allowing
 408  *      taskq_thread to continue execution.
 409  *
 410  *      For dynamic task queues, each bucket is marked as TQBUCKET_SUSPEND by
 411  *      taskq_suspend() function. After that taskq_bucket_dispatch() always
 412  *      fails, so that taskq_dispatch() will either enqueue tasks for a
 413  *      suspended backing queue or fail if TQ_NOQUEUE is specified in dispatch
 414  *      flags.
 415  *
 416  *      NOTE: taskq_suspend() does not immediately block any tasks already
 417  *            scheduled for dynamic task queues. It only suspends new tasks
 418  *            scheduled after taskq_suspend() was called.
 419  *
 420  *      taskq_member() function works by comparing a thread t_taskq pointer with
 421  *      the passed thread pointer.
 422  *
 423  * LOCKS and LOCK Hierarchy ----------------------------------------------------
 424  *
 425  *   There are three locks used in task queues:
 426  *
 427  *   1) The taskq_t's tq_lock, protecting global task queue state.
 428  *
 429  *   2) Each per-CPU bucket has a lock for bucket management.
 430  *
 431  *   3) The global taskq_cpupct_lock, which protects the list of
 432  *      TASKQ_THREADS_CPU_PCT taskqs.
 433  *
 434  *   If both (1) and (2) are needed, tq_lock should be taken *after* the bucket
 435  *   lock.
 436  *
 437  *   If both (1) and (3) are needed, tq_lock should be taken *after*
 438  *   taskq_cpupct_lock.
 439  *
 440  * DEBUG FACILITIES ------------------------------------------------------------
 441  *
 442  * For DEBUG kernels it is possible to induce random failures to
 443  * taskq_dispatch() function when it is given TQ_NOSLEEP argument. The value of
 444  * taskq_dmtbf and taskq_smtbf tunables control the mean time between induced
 445  * failures for dynamic and static task queues respectively.
 446  *
 447  * Setting TASKQ_STATISTIC to 0 will disable per-bucket statistics.
 448  *
 449  * TUNABLES --------------------------------------------------------------------
 450  *
 451  *      system_taskq_size       - Size of the global system_taskq.
 452  *                                This value is multiplied by nCPUs to determine
 453  *                                actual size.
```



```
 454  *                                Default value: 64
 455  *
 456  *      taskq_minimum_nthreads_max
 457  *                              - Minimum size of the thread list for a taskq.
 458  *                                Useful for testing different thread pool
 459  *                                sizes by overwriting tq_nthreads_target.
 460  *
 461  *      taskq_thread_timeout    - Maximum idle time for taskq_d_thread()
 462  *                                Default value: 5 minutes
 463  *
 464  *      taskq_maxbuckets        - Maximum number of buckets in any task queue
 465  *                                Default value: 128
 466  *
 467  *      taskq_search_depth      - Maximum # of buckets searched for a free entry
 468  *                                Default value: 4
 469  *
 470  *      taskq_dmtbf             - Mean time between induced dispatch failures
 471  *                                for dynamic task queues.
 472  *                                Default value: UINT_MAX (no induced failures)
 473  *
 474  *      taskq_smtbf             - Mean time between induced dispatch failures
 475  *                                for static task queues.
 476  *                                Default value: UINT_MAX (no induced failures)
 477  *
 478  * CONDITIONAL compilation -----------------------------------------------------
 479  *
 480  *    TASKQ_STATISTIC   - If set will enable bucket statistic (default).
 481  *
 482  */

 484 #include <sys/taskq_impl.h>
 485 #include <sys/thread.h>
 486 #include <sys/proc.h>
 487 #include <sys/kmem.h>
 488 #include <sys/vmem.h>
 489 #include <sys/callb.h>
 490 #include <sys/class.h>
 491 #include <sys/systm.h>
 492 #include <sys/cmn_err.h>
 493 #include <sys/debug.h>
 494 #include <sys/vmsystm.h>        /* For throttlefree */
 495 #include <sys/sysmacros.h>
 496 #include <sys/cpuvar.h>
 497 #include <sys/cpupart.h>
 498 #include <sys/sdt.h>
 499 #include <sys/sysdc.h>
 500 #include <sys/note.h>

 502 static kmem_cache_t *taskq_ent_cache, *taskq_cache;

 504 /*
 505  * Pseudo instance numbers for taskqs without explicitly provided instance.
 506  */
 507 static vmem_t *taskq_id_arena;

 509 /* Global system task queue for common use */
 510 taskq_t *system_taskq;

 512 /*
 513  * Maximum number of entries in global system taskq is
 514  *      system_taskq_size * max_ncpus
 515  */
 516 #define SYSTEM_TASKQ_SIZE 64
 517 int system_taskq_size = SYSTEM_TASKQ_SIZE;

 519 /*
```

```
 520  * Minimum size for tq_nthreads_max; useful for those who want to play around
 521  * with increasing a taskq's tq_nthreads_target.
 522  */
 523 int taskq_minimum_nthreads_max = 1;

 525 /*
 526  * We want to ensure that when taskq_create() returns, there is at least
 527  * one thread ready to handle requests.  To guarantee this, we have to wait
 528  * for the second thread, since the first one cannot process requests until
 529  * the second thread has been created.
 530  */
 531 #define TASKQ_CREATE_ACTIVE_THREADS     2

 533 /* Maximum percentage allowed for TASKQ_THREADS_CPU_PCT */
 534 #define TASKQ_CPUPCT_MAX_PERCENT        1000
 535 int taskq_cpupct_max_percent = TASKQ_CPUPCT_MAX_PERCENT;

 537 /*
 538  * Dynamic task queue threads that don't get any work within
 539  * taskq_thread_timeout destroy themselves
 540  */
 541 #define TASKQ_THREAD_TIMEOUT (60 * 5)
 542 int taskq_thread_timeout = TASKQ_THREAD_TIMEOUT;

 544 #define TASKQ_MAXBUCKETS 128
 545 int taskq_maxbuckets = TASKQ_MAXBUCKETS;

 547 /*
 548  * When a bucket has no available entries another buckets are tried.
 549  * taskq_search_depth parameter limits the amount of buckets that we search
 550  * before failing. This is mostly useful in systems with many CPUs where we may
 551  * spend too much time scanning busy buckets.
 552  */
 553 #define TASKQ_SEARCH_DEPTH 4
 554 int taskq_search_depth = TASKQ_SEARCH_DEPTH;

 556 /*
 557  * Hashing function: mix various bits of x. May be pretty much anything.
 558  */
 559 #define TQ_HASH(x) ((x) ^ ((x) >> 11) ^ ((x) >> 17) ^ ((x) ^ 27))

 561 /*
 562  * We do not create any new threads when the system is low on memory and start
 563  * throttling memory allocations. The following macro tries to estimate such
 564  * condition.
 565  */
 566 #define ENOUGH_MEMORY() (freemem > throttlefree)

 568 /*
 569  * Static functions.
 570  */
 571 static taskq_t  *taskq_create_common(const char *, int, int, pri_t, int,
 572     int, proc_t *, uint_t, uint_t);
 573 static void taskq_thread(void *);
 574 static void taskq_d_thread(taskq_ent_t *);
 575 static void taskq_bucket_extend(void *);
 576 static int  taskq_constructor(void *, void *, int);
 577 static void taskq_destructor(void *, void *);
 578 static int  taskq_ent_constructor(void *, void *, int);
 579 static void taskq_ent_destructor(void *, void *);
 580 static taskq_ent_t *taskq_ent_alloc(taskq_t *, int);
 581 static void taskq_ent_free(taskq_t *, taskq_ent_t *);
 582 static int taskq_ent_exists(taskq_t *, task_func_t, void *);
 583 static taskq_ent_t *taskq_bucket_dispatch(taskq_bucket_t *, task_func_t,
 584     void *);
```



```
 586 /*
 587  * Task queues kstats.
 588  */
 589 struct taskq_kstat {
 590         kstat_named_t   tq_pid;
 591         kstat_named_t   tq_tasks;
 592         kstat_named_t   tq_executed;
 593         kstat_named_t   tq_maxtasks;
 594         kstat_named_t   tq_totaltime;
 595         kstat_named_t   tq_nalloc;
 596         kstat_named_t   tq_nactive;
 597         kstat_named_t   tq_pri;
 598         kstat_named_t   tq_nthreads;
 599 } taskq_kstat = {
_____unchanged_portion_omitted_
```