NAME

taskqueue - asynchronous task execution

SYNOPSIS

#include <sys/param.h>
#include <sys/kernel.h>
#include <sys/malloc.h>
#include <sys/queue.h>
#include <sys/taskqueue.h>

```
typedef void (*task_fn_t)(void *context, int pending);

typedef void (*taskqueue_enqueue_fn)(void *context);

struct task {
        STAILQ_ENTRY(task)     ta_link;    /* link for queue */
        u_short                ta_pending;         /* count times queued */
        u_short                ta_priority;        /* priority of task in queue */
        task_fn_t        ta_func;  /* task handler */
        void                   *ta_context;        /* argument for handler */
};

enum taskqueue_callback_type {
        TASKQUEUE_CALLBACK_TYPE_INIT,
        TASKQUEUE_CALLBACK_TYPE_SHUTDOWN,
};

typedef void (*taskqueue_callback_fn)(void *context);

struct timeout_task;
```

*struct taskqueue **
taskqueue_create(*const char *name*, *int mflags*, *taskqueue_enqueue_fn enqueue*, *void *context*);

*struct taskqueue **
taskqueue_create_fast(*const char *name*, *int mflags*, *taskqueue_enqueue_fn enqueue*, *void *context*);

int
taskqueue_start_threads(*struct taskqueue **tqp*, *int count*, *int pri*, *const char *name*, *...*);

int

taskqueue_start_threads_cpuset(*struct taskqueue **tqp*, *int count*, *int pri*, *cpuset_t *mask*,
*const char *name*, *...*);

int
taskqueue_start_threads_in_proc(*struct taskqueue **tqp*, *int count*, *int pri*, *struct proc *proc*,
*const char *name*, *...*);

void
taskqueue_set_callback(*struct taskqueue *queue*, *enum taskqueue_callback_type cb_type*,
taskqueue_callback_fn callback, *void *context*);

void
taskqueue_free(*struct taskqueue *queue*);

int
taskqueue_enqueue(*struct taskqueue *queue*, *struct task *task*);

int
taskqueue_enqueue_flags(*struct taskqueue *queue*, *struct task *task*, *int flags*);

int
taskqueue_enqueue_timeout(*struct taskqueue *queue*, *struct timeout_task *timeout_task*, *int ticks*);

int
taskqueue_enqueue_timeout_sbt(*struct taskqueue *queue*, *struct timeout_task *timeout_task*,
sbintime_t sbt, *sbintime_t pr*, *int flags*);

int
taskqueue_cancel(*struct taskqueue *queue*, *struct task *task*, *u_int *pendp*);

int
taskqueue_cancel_timeout(*struct taskqueue *queue*, *struct timeout_task *timeout_task*, *u_int *pendp*);

void
taskqueue_drain(*struct taskqueue *queue*, *struct task *task*);

void
taskqueue_drain_timeout(*struct taskqueue *queue*, *struct timeout_task *timeout_task*);

void

taskqueue_drain_all(*struct taskqueue *queue*);

void
taskqueue_quiesce(*struct taskqueue *queue*);

void
taskqueue_block(*struct taskqueue *queue*);

void
taskqueue_unblock(*struct taskqueue *queue*);

int
taskqueue_member(*struct taskqueue *queue*, *struct thread *td*);

void
taskqueue_run(*struct taskqueue *queue*);

TASK_INIT(*struct task *task*, *int priority*, *task_fn_t func*, *void *context*);

TASK_INITIALIZER(*int priority*, *task_fn_t func*, *void *context*);

TASKQUEUE_DECLARE(*name*);

TASKQUEUE_DEFINE(*name*, *taskqueue_enqueue_fn enqueue*, *void *context*, *init*);

TASKQUEUE_FAST_DEFINE(*name*, *taskqueue_enqueue_fn enqueue*, *void *context*, *init*);

TASKQUEUE_DEFINE_THREAD(*name*);

TASKQUEUE_FAST_DEFINE_THREAD(*name*);

TIMEOUT_TASK_INIT(*struct taskqueue *queue*, *struct timeout_task *timeout_task*, *int priority*, *task_fn_t func*, *void *context*);

DESCRIPTION

These functions provide a simple interface for asynchronous execution of code.

The function **taskqueue_create**() is used to create new queues. The arguments to **taskqueue_create**() include a name that should be unique, a set of malloc(9) flags that specify whether the call to **malloc**() is allowed to sleep, a function that is called from **taskqueue_enqueue**() when a task is added to the queue, and a pointer to the memory location where the identity of the thread that services the queue is recorded. The function called from **taskqueue_enqueue**() must arrange for the queue to be processed (for instance by scheduling a software interrupt or waking a kernel thread). The memory location where the thread identity is recorded is used to signal the service thread(s) to terminate--when this value is set to zero and the thread is signaled it will terminate. If the queue is intended for use in fast interrupt handlers **taskqueue_create_fast**() should be used in place of **taskqueue_create**().

The function **taskqueue_free**() should be used to free the memory used by the queue. Any tasks that are on the queue will be executed at this time after which the thread servicing the queue will be signaled that it should exit.

Once a taskqueue has been created, its threads should be started using **taskqueue_start_threads**(), **taskqueue_start_threads_cpuset**() or **taskqueue_start_threads_in_proc**(). **taskqueue_start_threads_cpuset**() takes a *cpuset* argument which will cause the threads which are started for the taskqueue to be restricted to run on the given CPUs. **taskqueue_start_threads_in_proc**() takes a *proc* argument which will cause the threads which are started for the taskqueue to be assigned to the given kernel process. Callbacks may optionally be registered using **taskqueue_set_callback**(). Currently, callbacks may be registered for the following purposes:

TASKQUEUE_CALLBACK_TYPE_INIT
This callback is called by every thread in the taskqueue, before it executes any tasks. This callback must be set before the taskqueue's threads are started.

TASKQUEUE_CALLBACK_TYPE_SHUTDOWN
This callback is called by every thread in the taskqueue, after it executes its last task. This callback will always be called before the taskqueue structure is reclaimed.

To add a task to the list of tasks queued on a taskqueue, call **taskqueue_enqueue**() with pointers to the queue and task. If the task's *ta_pending* field is non-zero, then it is simply incremented to reflect the number of times the task was enqueued, up to a cap of USHRT_MAX. Otherwise, the task is added to the list before the first task which has a lower *ta_priority* value or at the end of the list if no tasks have a lower priority. Enqueueing a task does not perform any memory allocation which makes it suitable for calling from an interrupt handler. This function will return EPIPE if the queue is being freed.

When a task is executed, first it is removed from the queue, the value of *ta_pending* is recorded and then the field is zeroed. The function *ta_func* from the task structure is called with the value of the field *ta_context* as its first argument and the value of *ta_pending* as its second argument. After the function *ta_func* returns, wakeup(9) is called on the task pointer passed to **taskqueue_enqueue**().

The **taskqueue_enqueue_flags**() accepts an extra *flags* parameter which specifies a set of optional flags to alter the behavior of **taskqueue_enqueue**(). It contains one or more of the following flags:

TASKQUEUE_FAIL_IF_PENDING
: **taskqueue_enqueue_flags**() fails if the task is already scheduled for execution. EEXIST is returned and the *ta_pending* counter value remains unchanged.

TASKQUEUE_FAIL_IF_CANCELING
: **taskqueue_enqueue_flags**() fails if the task is in the canceling state and ECANCELED is returned.

The **taskqueue_enqueue_timeout**() function is used to schedule the enqueue after the specified number of *ticks*. The **taskqueue_enqueue_timeout_sbt**() function provides finer control over the scheduling based on *sbt*, *pr*, and *flags*, as detailed in callout(9). If the *ticks* argument is negative, the already scheduled enqueueing is not re-scheduled. Otherwise, the task is scheduled for enqueueing in the future, after the absolute value of *ticks* is passed. This function returns -1 if the task is being drained. Otherwise, the number of pending calls is returned.

The **taskqueue_cancel**() function is used to cancel a task. The *ta_pending* count is cleared, and the old value returned in the reference parameter *pendp*, if it is non-NULL. If the task is currently running, EBUSY is returned, otherwise 0. To implement a blocking **taskqueue_cancel**() that waits for a running task to finish, it could look like:

```
while (taskqueue_cancel(tq, task, NULL) != 0)
        taskqueue_drain(tq, task);
```

Note that, as with **taskqueue_drain**(), the caller is responsible for ensuring that the task is not re-enqueued after being canceled.

Similarly, the **taskqueue_cancel_timeout**() function is used to cancel the scheduled task execution.

The **taskqueue_drain**() function is used to wait for the task to finish, and the **taskqueue_drain_timeout**() function is used to wait for the scheduled task to finish. There is no guarantee that the task will not be enqueued after call to **taskqueue_drain**(). If the caller wants to put the task into a known state, then before calling **taskqueue_drain**() the caller should use out-of-band means to ensure that the task would not be enqueued. For example, if the task is enqueued by an interrupt filter, then the interrupt could be disabled.

The **taskqueue_drain_all**() function is used to wait for all pending and running tasks that are enqueued on the taskqueue to finish. Tasks posted to the taskqueue after **taskqueue_drain_all**() begins processing, including pending enqueues scheduled by a previous call to **taskqueue_enqueue_timeout**(), do not extend the wait time of **taskqueue_drain_all**() and may complete after **taskqueue_drain_all**() returns. The **taskqueue_quiesce**() function is used to wait for the queue to become empty and for all running tasks to finish. To avoid blocking indefinitely, the caller must ensure by some mechanism that tasks will eventually stop being posted to the queue.

The **taskqueue_block**() function blocks the taskqueue. It prevents any enqueued but not running tasks from being executed. Future calls to **taskqueue_enqueue**() will enqueue tasks, but the tasks will not be run until **taskqueue_unblock**() is called. Please note that **taskqueue_block**() does not wait for any currently running tasks to finish. Thus, the **taskqueue_block**() does not provide a guarantee that **taskqueue_run**() is not running after **taskqueue_block**() returns, but it does provide a guarantee that **taskqueue_run**() will not be called again until **taskqueue_unblock**() is called. If the caller requires a guarantee that **taskqueue_run**() is not running, then this must be arranged by the caller. Note that if **taskqueue_drain**() is called on a task that is enqueued on a taskqueue that is blocked by **taskqueue_block**(), then **taskqueue_drain**() can not return until the taskqueue is unblocked. This can result in a deadlock if the thread blocked in **taskqueue_drain**() is the thread that is supposed to call **taskqueue_unblock**(). Thus, use of **taskqueue_drain**() after **taskqueue_block**() is discouraged, because the state of the task can not be known in advance. The same caveat applies to **taskqueue_drain_all**().

The **taskqueue_unblock**() function unblocks the previously blocked taskqueue. All enqueued tasks can be run after this call.

The **taskqueue_member**() function returns 1 if the given thread *td* is part of the given taskqueue *queue* and 0 otherwise.

The **taskqueue_run**() function will run all pending tasks in the specified *queue*. Normally this function is only used internally.

A convenience macro, **TASK_INIT**(*task*, *priority*, *func*, *context*) is provided to initialise a *task* structure. The **TASK_INITIALIZER**() macro generates an initializer for a task structure. A macro **TIMEOUT_TASK_INIT**(*queue*, *timeout_task*, *priority*, *func*, *context*) initializes the *timeout_task* structure. The values of *priority*, *func*, and *context* are simply copied into the task structure fields and the *ta_pending* field is cleared.

Five macros **TASKQUEUE_DECLARE**(*name*), **TASKQUEUE_DEFINE**(*name*, *enqueue*, *context*, *init*), **TASKQUEUE_FAST_DEFINE**(*name*, *enqueue*, *context*, *init*), and **TASKQUEUE_DEFINE_THREAD**(*name*) **TASKQUEUE_FAST_DEFINE_THREAD**(*name*) are used to declare a reference to a global queue, to define the implementation of the queue, and declare a queue that uses its own thread. The **TASKQUEUE_DEFINE**() macro arranges to call **taskqueue_create**() with the values of its *name*, *enqueue* and *context* arguments during system initialisation. After calling **taskqueue_create**(), the *init* argument to the macro is executed as a C statement, allowing any further initialisation to be performed (such as registering an interrupt handler, etc.).

The **TASKQUEUE_DEFINE_THREAD**() macro defines a new taskqueue with its own kernel thread to serve tasks. The variable *struct taskqueue *taskqueue_name* is used to enqueue tasks onto the queue.

TASKQUEUE_FAST_DEFINE() and **TASKQUEUE_FAST_DEFINE_THREAD**() act just like **TASKQUEUE_DEFINE**() and **TASKQUEUE_DEFINE_THREAD**() respectively but taskqueue is created with **taskqueue_create_fast**().

Predefined Task Queues

The system provides four global taskqueues, *taskqueue_fast*, *taskqueue_swi*, *taskqueue_swi_giant*, and *taskqueue_thread*. The *taskqueue_fast* queue is for swi handlers dispatched from fast interrupt handlers, where sleep mutexes cannot be used. The swi taskqueues are run via a software interrupt mechanism. The *taskqueue_swi* queue runs without the protection of the *Giant* kernel lock, and the *taskqueue_swi_giant* queue runs with the protection of the *Giant* kernel lock. The thread taskqueue *taskqueue_thread* runs in a kernel thread context, and tasks run from this thread do not run under the *Giant* kernel lock. If the caller wants to run under *Giant*, he should explicitly acquire and release *Giant* in his taskqueue handler routine.

To use these queues, call **taskqueue_enqueue**() with the value of the global taskqueue variable for the queue you wish to use.

The software interrupt queues can be used, for instance, for implementing interrupt handlers which must perform a significant amount of processing in the handler. The hardware interrupt handler would perform minimal processing of the interrupt and then enqueue a task to finish the work. This reduces to a minimum the amount of time spent with interrupts disabled.

The thread queue can be used, for instance, by interrupt level routines that need to call kernel functions that do things that can only be done from a thread context. (e.g., call malloc with the M_WAITOK flag.)

Note that tasks queued on shared taskqueues such as *taskqueue_swi* may be delayed an indeterminate amount of time before execution. If queueing delays cannot be tolerated then a private taskqueue should be created with a dedicated processing thread.

SEE ALSO

callout(9), ithread(9), kthread(9), swi(9)

HISTORY

This interface first appeared in FreeBSD 5.0. There is a similar facility called work_queue in the Linux kernel.

AUTHORS

This manual page was written by Doug Rabson.