NAME

callout_active, **callout_deactivate**, **callout_async_drain**, **callout_drain**, **callout_init**, **callout_init_mtx**, **callout_init_rm**, **callout_init_rw**, **callout_pending**, **callout_reset**, **callout_reset_curcpu**, **callout_reset_on**, **callout_reset_sbt**, **callout_reset_sbt_curcpu**, **callout_reset_sbt_on**, **callout_schedule**, **callout_schedule_curcpu**, **callout_schedule_on**, **callout_schedule_sbt**, **callout_schedule_sbt_curcpu**, **callout_schedule_sbt_on**, **callout_stop**, **callout_when** - execute a function after a specified length of time

SYNOPSIS

#include <sys/types.h>
#include <sys/callout.h>

typedef void callout_func_t (void *);
int
callout_active(*struct callout *c*);

void
callout_deactivate(*struct callout *c*);

int
callout_async_drain(*struct callout *c*, *callout_func_t *drain*);

int
callout_drain(*struct callout *c*);

void
callout_init(*struct callout *c*, *int mpsafe*);

void
callout_init_mtx(*struct callout *c*, *struct mtx *mtx*, *int flags*);

void
callout_init_rm(*struct callout *c*, *struct rmlock *rm*, *int flags*);

void
callout_init_rw(*struct callout *c*, *struct rwlock *rw*, *int flags*);

int
callout_pending(*struct callout *c*);

int

callout_reset(*struct callout *c, int ticks, callout_func_t *func, void *arg*);

int
callout_reset_curcpu(*struct callout *c, int ticks, callout_func_t *func, void *arg*);

int
callout_reset_on(*struct callout *c, int ticks, callout_func_t *func, void *arg, int cpu*);

int
callout_reset_sbt(*struct callout *c, sbintime_t sbt, sbintime_t pr, callout_func_t *func, void *arg, int flags*);

int
callout_reset_sbt_curcpu(*struct callout *c, sbintime_t sbt, sbintime_t pr, callout_func_t *func, void *arg, int flags*);

int
callout_reset_sbt_on(*struct callout *c, sbintime_t sbt, sbintime_t pr, callout_func_t *func, void *arg, int cpu, int flags*);

int
callout_schedule(*struct callout *c, int ticks*);

int
callout_schedule_curcpu(*struct callout *c, int ticks*);

int
callout_schedule_on(*struct callout *c, int ticks, int cpu*);

int
callout_schedule_sbt(*struct callout *c, sbintime_t sbt, sbintime_t pr, int flags*);

int
callout_schedule_sbt_curcpu(*struct callout *c, sbintime_t sbt, sbintime_t pr, int flags*);

int
callout_schedule_sbt_on(*struct callout *c, sbintime_t sbt, sbintime_t pr, int cpu, int flags*);

int
callout_stop(*struct callout *c*);

sbintime_t
callout_when(*sbintime_t sbt, sbintime_t precision, int flags, sbintime_t *sbt_res, sbintime_t *precision_res*);

DESCRIPTION

The **callout** API is used to schedule a call to an arbitrary function at a specific time in the future. Consumers of this API are required to allocate a callout structure (struct callout) for each pending function invocation. This structure stores state about the pending function invocation including the function to be called and the time at which the function should be invoked. Pending function calls can be cancelled or rescheduled to a different time. In addition, a callout structure may be reused to schedule a new function call after a scheduled call is completed.

Callouts only provide a single-shot mode. If a consumer requires a periodic timer, it must explicitly reschedule each function call. This is normally done by rescheduling the subsequent call within the called function.

Callout functions must not sleep. They may not acquire sleepable locks, wait on condition variables, perform blocking allocation requests, or invoke any other action that might sleep.

Each callout structure must be initialized by **callout_init**(), **callout_init_mtx**(), **callout_init_rm**(), or **callout_init_rw**() before it is passed to any of the other callout functions. The **callout_init**() function initializes a callout structure in *c* that is not associated with a specific lock. If the *mpsafe* argument is zero, the callout structure is not considered to be "multi-processor safe"; and the Giant lock will be acquired before calling the callout function and released when the callout function returns.

The **callout_init_mtx**(), **callout_init_rm**(), and **callout_init_rw**() functions initialize a callout structure in *c* that is associated with a specific lock. The lock is specified by the *mtx*, *rm*, or *rw* parameter. The associated lock must be held while stopping or rescheduling the callout. The callout subsystem acquires the associated lock before calling the callout function and releases it after the function returns. If the callout was cancelled while the callout subsystem waited for the associated lock, the callout function is not called, and the associated lock is released. This ensures that stopping or rescheduling the callout will abort any previously scheduled invocation.

A sleepable read-mostly lock (one initialized with the RM_SLEEPABLE flag) may not be used with **callout_init_rm**(). Similarly, other sleepable lock types such as sx(9) and lockmgr(9) cannot be used with callouts because sleeping is not permitted in the callout subsystem.

These *flags* may be specified for **callout_init_mtx**(), **callout_init_rm**(), or **callout_init_rw**():

CALLOUT_RETURNUNLOCKED The callout function will release the associated lock itself, so the

callout subsystem should not attempt to unlock it after the callout function returns.

CALLOUT_SHAREDLOCK The lock is only acquired in read mode when running the callout handler. This flag is ignored by **callout_init_mtx**().

The function **callout_stop**() cancels a callout *c* if it is currently pending. If the callout is pending and successfully stopped, then **callout_stop**() returns a value of one. If the callout is not set, or has already been serviced, then negative one is returned. If the callout is currently being serviced and cannot be stopped, then zero will be returned. If the callout is currently being serviced and cannot be stopped, and at the same time a next invocation of the same callout is also scheduled, then **callout_stop**() unschedules the next run and returns zero. If the callout has an associated lock, then that lock must be held when this function is called.

The function **callout_async_drain**() is identical to **callout_stop**() with one difference. When **callout_async_drain**() returns zero it will arrange for the function *drain* to be called using the same argument given to the **callout_reset**() function. **callout_async_drain**() If the callout has an associated lock, then that lock must be held when this function is called. Note that when stopping multiple callouts that use the same lock it is possible to get multiple return's of zero and multiple calls to the *drain* function, depending upon which CPU's the callouts are running. The *drain* function itself is called from the context of the completing callout i.e. softclock or hardclock, just like a callout itself.

The function **callout_drain**() is identical to **callout_stop**() except that it will wait for the callout *c* to complete if it is already in progress. This function MUST NOT be called while holding any locks on which the callout might block, or deadlock will result. Note that if the callout subsystem has already begun processing this callout, then the callout function may be invoked before **callout_drain**() returns. However, the callout subsystem does guarantee that the callout will be fully stopped before **callout_drain**() returns.

The **callout_reset**() and **callout_schedule**() function families schedule a future function invocation for callout *c*. If *c* already has a pending callout, it is cancelled before the new invocation is scheduled. These functions return a value of one if a pending callout was cancelled and zero if there was no pending callout. If the callout has an associated lock, then that lock must be held when any of these functions are called.

The time at which the callout function will be invoked is determined by either the *ticks* argument or the *sbt*, *pr*, and *flags* arguments. When *ticks* is used, the callout is scheduled to execute after *ticks*/hz seconds. Non-positive values of *ticks* are silently converted to the value '1'.

The *sbt*, *pr*, and *flags* arguments provide more control over the scheduled time including support for

higher resolution times, specifying the precision of the scheduled time, and setting an absolute deadline instead of a relative timeout. The callout is scheduled to execute in a time window which begins at the time specified in *sbt* and extends for the amount of time specified in *pr*. If *sbt* specifies a time in the past, the window is adjusted to start at the current time. A non-zero value for *pr* allows the callout subsystem to coalesce callouts scheduled close to each other into fewer timer interrupts, reducing processing overhead and power consumption. These *flags* may be specified to adjust the interpretation of *sbt* and *pr*:

C_ABSOLUTE Handle the *sbt* argument as an absolute time since boot. By default, *sbt* is treated as a relative amount of time, similar to *ticks*.

C_DIRECT_EXEC Run the handler directly from hardware interrupt context instead of from the softclock thread. This reduces latency and overhead, but puts more constraints on the callout function. Callout functions run in this context may use only spin mutexes for locking and should be as small as possible because they run with absolute priority.

C_PREL() Specifies relative event time precision as binary logarithm of time interval divided by acceptable time deviation: 1 -- 1/2, 2 -- 1/4, etc. Note that the larger of *pr* or this value is used as the length of the time window. Smaller values (which result in larger time intervals) allow the callout subsystem to aggregate more events in one timer interrupt.

C_PRECALC The *sbt* argument specifies the absolute time at which the callout should be run, and the *pr* argument specifies the requested precision, which will not be adjusted during the scheduling process. The *sbt* and *pr* values should be calculated by an earlier call to **callout_when**() which uses the user-supplied *sbt*, *pr*, and *flags* values.

C_HARDCLOCK Align the timeouts to **hardclock**() calls if possible.

The **callout_reset**() functions accept a *func* argument which identifies the function to be called when the time expires. It must be a pointer to a function that takes a single *void* * argument. Upon invocation, *func* will receive *arg* as its only argument. The **callout_schedule**() functions reuse the *func* and *arg* arguments from the previous callout. Note that one of the **callout_reset**() functions must always be called to initialize *func* and *arg* before one of the **callout_schedule**() functions can be used.

The callout subsystem provides a softclock thread for each CPU in the system. Callouts are assigned to a single CPU and are executed by the softclock thread for that CPU. Initially, callouts are assigned to CPU 0. The **callout_reset_on**(), **callout_reset_sbt_on**(), **callout_schedule_on**() and **callout_schedule_sbt_on**() functions assign the callout to CPU *cpu*. The **callout_reset_curcpu**(),

callout_reset_sbt_curpu(), **callout_schedule_curcpu**() and **callout_schedule_sbt_curcpu**() functions assign the callout to the current CPU. The **callout_reset**(), **callout_reset_sbt**(), **callout_schedule**() and **callout_schedule_sbt**() functions schedule the callout to execute in the softclock thread of the CPU to which it is currently assigned.

Softclock threads are not pinned to their respective CPUs by default. The softclock thread for CPU 0 can be pinned to CPU 0 by setting the *kern.pin_default_swi* loader tunable to a non-zero value. Softclock threads for CPUs other than zero can be pinned to their respective CPUs by setting the *kern.pin_pcpu_swi* loader tunable to a non-zero value.

The macros **callout_pending**(), **callout_active**() and **callout_deactivate**() provide access to the current state of the callout. The **callout_pending**() macro checks whether a callout is *pending*; a callout is considered *pending* when a timeout has been set but the time has not yet arrived. Note that once the timeout time arrives and the callout subsystem starts to process this callout, **callout_pending**() will return FALSE even though the callout function may not have finished (or even begun) executing. The **callout_active**() macro checks whether a callout is marked as *active*, and the **callout_deactivate**() macro clears the callout's *active* flag. The callout subsystem marks a callout as *active* when a timeout is set and it clears the *active* flag in **callout_stop**() and **callout_drain**(), but it *does not* clear it when a callout expires normally via the execution of the callout function.

The **callout_when**() function may be used to pre-calculate the absolute time at which the timeout should be run and the precision of the scheduled run time according to the required time *sbt*, precision *precision*, and additional adjustments requested by the *flags* argument. Flags accepted by the **callout_when**() function are the same as flags for the **callout_reset**() function. The resulting time is assigned to the variable pointed to by the *sbt_res* argument, and the resulting precision is assigned to **precision_res*. When passing the results to *callout_reset*, add the *C_PRECALC* flag to *flags*, to avoid incorrect re-adjustment. The function is intended for situations where precise time of the callout run should be known in advance, since trying to read this time from the callout structure itself after a **callout_reset**() call is racy.

Avoiding Race Conditions

The callout subsystem invokes callout functions from its own thread context. Without some kind of synchronization, it is possible that a callout function will be invoked concurrently with an attempt to stop or reset the callout by another thread. In particular, since callout functions typically acquire a lock as their first action, the callout function may have already been invoked, but is blocked waiting for that lock at the time that another thread tries to reset or stop the callout.

There are three main techniques for addressing these synchronization concerns. The first approach is preferred as it is the simplest:

1. Callouts can be associated with a specific lock when they are initialized by **callout_init_mtx**(), **callout_init_rm**(), or **callout_init_rw**(). When a callout is associated with a lock, the callout subsystem acquires the lock before the callout function is invoked. This allows the callout subsystem to transparently handle races between callout cancellation, scheduling, and execution. Note that the associated lock must be acquired before calling **callout_stop**() or one of the **callout_reset**() or **callout_schedule**() functions to provide this safety.

   A callout initialized via **callout_init**() with *mpsafe* set to zero is implicitly associated with the *Giant* mutex. If *Giant* is held when cancelling or rescheduling the callout, then its use will prevent races with the callout function.

2. The return value from **callout_stop**() (or the **callout_reset**() and **callout_schedule**() function families) indicates whether or not the callout was removed. If it is known that the callout was set and the callout function has not yet executed, then a return value of FALSE indicates that the callout function is about to be called. For example:

```
if (sc->sc_flags & SCFLG_CALLOUT_RUNNING) {
        if (callout_stop(&sc->sc_callout)) {
                sc->sc_flags &= ~SCFLG_CALLOUT_RUNNING;
                /* successfully stopped */
        } else {
                /*
                 * callout has expired and callout
                 * function is about to be executed
                 */
        }
}
```

3. The **callout_pending**(), **callout_active**() and **callout_deactivate**() macros can be used together to work around the race conditions. When a callout's timeout is set, the callout subsystem marks the callout as both *active* and *pending*. When the timeout time arrives, the callout subsystem begins processing the callout by first clearing the *pending* flag. It then invokes the callout function without changing the *active* flag, and does not clear the *active* flag even after the callout function returns. The mechanism described here requires the callout function itself to clear the *active* flag using the **callout_deactivate**() macro. The **callout_stop**() and **callout_drain**() functions always clear both the *active* and *pending* flags before returning.

   The callout function should first check the *pending* flag and return without action if **callout_pending**() returns TRUE. This indicates that the callout was rescheduled using

callout_reset() just before the callout function was invoked. If **callout_active**() returns FALSE then the callout function should also return without action. This indicates that the callout has been stopped. Finally, the callout function should call **callout_deactivate**() to clear the *active* flag. For example:

```
mtx_lock(&sc->sc_mtx);
if (callout_pending(&sc->sc_callout)) {
        /* callout was reset */
        mtx_unlock(&sc->sc_mtx);
        return;
}
if (!callout_active(&sc->sc_callout)) {
        /* callout was stopped */
        mtx_unlock(&sc->sc_mtx);
        return;
}
callout_deactivate(&sc->sc_callout);
/* rest of callout function */
```

Together with appropriate synchronization, such as the mutex used above, this approach permits the **callout_stop**() and **callout_reset**() functions to be used at any time without races. For example:

```
mtx_lock(&sc->sc_mtx);
callout_stop(&sc->sc_callout);
/* The callout is effectively stopped now. */
```

If the callout is still pending then these functions operate normally, but if processing of the callout has already begun then the tests in the callout function cause it to return without further action. Synchronization between the callout function and other code ensures that stopping or resetting the callout will never be attempted while the callout function is past the **callout_deactivate**() call.

The above technique additionally ensures that the *active* flag always reflects whether the callout is effectively enabled or disabled. If **callout_active**() returns false, then the callout is effectively disabled, since even if the callout subsystem is actually just about to invoke the callout function, the callout function will return without action.

There is one final race condition that must be considered when a callout is being stopped for the last time. In this case it may not be safe to let the callout function itself detect that the callout was stopped,

since it may need to access data objects that have already been destroyed or recycled. To ensure that the callout is completely finished, a call to **callout_drain**() should be used. In particular, a callout should always be drained prior to destroying its associated lock or releasing the storage for the callout structure.

RETURN VALUES

The **callout_active**() macro returns the state of a callout's *active* flag.

The **callout_pending**() macro returns the state of a callout's *pending* flag.

The **callout_reset**() and **callout_schedule**() function families return a value of one if the callout was pending before the new function invocation was scheduled.

The **callout_stop**() and **callout_drain**() functions return a value of one if the callout was still pending when it was called, a zero if the callout could not be stopped and a negative one is it was either not running or has already completed.

HISTORY

FreeBSD initially used the long standing BSD linked list callout mechanism which offered O(n) insertion and removal running time but did not generate or require handles for untimeout operations.

FreeBSD 3.0 introduced a new set of timeout and untimeout routines from NetBSD based on the work of Adam M. Costello and George Varghese, published in a technical report entitled *Redesigning the BSD Callout and Timer Facilities* and modified for inclusion in FreeBSD by Justin T. Gibbs. The original work on the data structures used in that implementation was published by G. Varghese and A. Lauck in the paper *Hashed and Hierarchical Timing Wheels: Data Structures for the Efficient Implementation of a Timer Facility* in the *Proceedings of the 11th ACM Annual Symposium on Operating Systems Principles.*

FreeBSD 3.3 introduced the first implementations of **callout_init**(), **callout_reset**(), and **callout_stop**() which permitted callers to allocate dedicated storage for callouts. This ensured that a callout would always fire unlike **timeout**() which would silently fail if it was unable to allocate a callout.

FreeBSD 5.0 permitted callout handlers to be tagged as MPSAFE via **callout_init**().

FreeBSD 5.3 introduced **callout_drain**().

FreeBSD 6.0 introduced **callout_init_mtx**().

FreeBSD 8.0 introduced per-CPU callout wheels, **callout_init_rw**(), and **callout_schedule**().

FreeBSD 9.0 changed the underlying timer interrupts used to drive callouts to prefer one-shot event timers instead of a periodic timer interrupt.

FreeBSD 10.0 switched the callout wheel to support tickless operation. These changes introduced *sbintime_t* and the **callout_reset_sbt***() family of functions. FreeBSD 10.0 also added C_DIRECT_EXEC and **callout_init_rm**().

FreeBSD 10.2 introduced the **callout_schedule_sbt***() family of functions.

FreeBSD 11.0 introduced **callout_async_drain**(). FreeBSD 11.1 introduced **callout_when**(). FreeBSD 13.0 removed *timeout_t*, **timeout**(), and **untimeout**().