NAME

intr_event_add_handler, **intr_event_create**, **intr_event_destroy**, **intr_event_handle**, **intr_event_remove_handler**, **intr_priority** - kernel interrupt handler and thread API

SYNOPSIS

#include <sys/param.h>
#include <sys/bus.h>
#include <sys/interrupt.h>

int
intr_event_add_handler(*struct intr_event *ie, const char *name, driver_filter_t filter, driver_intr_t handler, void *arg, u_char pri, enum intr_type flags, void **cookiep*);

int
intr_event_create(*struct intr_event **event, void *source, int flags, int irq, void (*pre_ithread)(void *), void (*post_ithread)(void *), void (*post_filter)(void *), int (*assign_cpu)(void *, int), const char *fmt, ...*);

int
intr_event_destroy(*struct intr_event *ie*);

int
intr_event_handle(*struct intr_event *ie, struct trapframe *frame*);

int
intr_event_remove_handler(*void *cookie*);

u_char
intr_priority(*enum intr_type flags*);

DESCRIPTION

The interrupt event API provides methods to manage the registration and execution of interrupt handlers and their associated thread contexts.

Each interrupt event in the system corresponds to a single hardware or software interrupt source. Each interrupt event maintains a list of interrupt handlers, sorted by priority, which will be invoked when handling the event. An interrupt event will typically, but not always, have an associated kthread(9), known as the interrupt thread. Finally, each event contains optional callback functions which will be invoked before and after the handler functions themselves.

An interrupt handler contains two distinct handler functions: the *filter* and the thread *handler*. The *filter* function is run from interrupt context and is intended to perform quick handling such as acknowledging or masking a hardware interrupt, and queueing work for the ensuing thread *handler*. Both functions are optional; each interrupt handler may choose to register a filter, a thread handler, or both. Each interrupt handler also consists of a name, a set of flags, and an opaque argument which will be passed to both the *filter* and *handler* functions.

Handler Constraints

The *filter* function is executed inside a critical(9) section. Therefore, filters may not yield the CPU for any reason, and may only use spin locks to access shared data. Allocating memory within a filter is not permitted.

The *handler* function executes from the context of the associated interrupt kernel thread. Sleeping is not permitted, but the interrupt thread may be preempted by higher priority threads. Thus, threaded handler functions may obtain non-sleepable locks, as described in locking(9). Any memory or zone allocations in an interrupt thread must specify the M_NOWAIT flag, and any allocation errors must be handled.

The exception to these constraints is software interrupt threads, which are allowed to sleep but should be allocated and scheduled using the swi(9) interface.

Function Descriptions

The **intr_event_create**() function creates a new interrupt event. The *event* argument points to a *struct intr_event* pointer that will reference the newly created event upon success. The *source* argument is an opaque pointer which will be passed to the *pre_ithread*, *post_ithread*, and *post_filter* callbacks. The *flags* argument is a mask of properties of this thread. The only valid flag currently for **intr_event_create**() is IE_SOFT to specify that this interrupt thread is a software interrupt. The *enable* and *disable* arguments specify optional functions used to enable and disable this interrupt thread's interrupt source. The *irq* argument is the unique interrupt vector number corresponding to the event. The *pre_ithread*, *post_ithread*, and *post_filter* arguments are callback functions that are invoked at different points while handling an interrupt. This is described in more detail in the *Handler Callbacks* section, below. They may be *NULL* to specify no callback. The *assign_cpu* argument points to a callback function that will be invoked when binding an interrupt to a particular CPU. It may be *NULL* if binding is unsupported. The remaining arguments form a printf(9) argument list that is used to build the base name of the new interrupt thread. The full name of an interrupt thread is formed by concatenating the base name of the interrupt thread with the names of all of its interrupt handlers.

The **intr_event_destroy**() function destroys a previously created interrupt event by releasing its resources. An interrupt event can only be destroyed if it has no handlers remaining.

The **intr_event_add_handler**() function adds a new handler to an existing interrupt event specified by *ie*.

The *name* argument specifies a name for this handler. The *filter* argument provide the filter function to execute. The *handler* argument provides the handler function to be executed from the event's interrupt thread. The *arg* argument will be passed to the *filter* and *handler* functions when they are invoked. The *pri* argument specifies the priority of this handler, corresponding to the values defined in <*sys/priority.h*>. It determines the order this handler is called relative to the other handlers for this event, as well as the scheduling priority of of the backing kernel thread. *flags* argument can be used to specify properties of this handler as defined in <*sys/bus.h*>. If *cookiep* is not NULL, then it will be assigned a cookie that can be used later to remove this handler.

The **intr_event_handle**() function is the main entry point into the interrupt handling code. It must be called from an interrupt context. The function will execute all filter handlers associated with the interrupt event *ie*, and schedule the associated interrupt thread to run, if applicable. The *frame* argument is used to pass a pointer to the *struct trapframe* containing the machine state at the time of the interrupt. The main body of this function runs within a critical(9) section.

The **intr_event_remove_handler**() function removes an interrupt handler from the interrupt event specified by *ie*. The *cookie* argument, obtained from **intr_event_add_handler**(), identifies the handler to remove.

The **intr_priority**() function translates the INTR_TYPE_* interrupt flags into interrupt thread scheduling priorities.

The interrupt flags not related to the type of a particular interrupt (INTR_TYPE_*) can be used to specify additional properties of both hardware and software interrupt handlers. The INTR_EXCL flag specifies that this handler cannot share an interrupt thread with another handler. The INTR_MPSAFE flag specifies that this handler is MP safe in that it does not need the Giant mutex to be held while it is executed. The INTR_ENTROPY flag specifies that the interrupt source this handler is tied to is a good source of entropy, and thus that entropy should be gathered when an interrupt from the handler's source triggers. Presently, the INTR_ENTROPY flag is not valid for software interrupt handlers.

Handler Callbacks

Each *struct intr_event* is assigned three optional callback functions when it is created: *pre_ithread*, *post_ithread*, and *post_filter*. These callbacks are intended to be defined by the interrupt controller driver, to allow for actions such as masking and unmasking hardware interrupt signals.

When an interrupt is triggered, all filters are run to determine if any threaded interrupt handlers should be scheduled for execution by the associated interrupt thread. If no threaded handlers are scheduled, the *post_filter* callback is invoked which should acknowledge the interrupt and permit it to trigger in the future. If any threaded handlers are scheduled, the *pre_ithread* callback is invoked instead. This handler should acknowledge the interrupt, but it should also ensure that the interrupt will not fire continuously

until after the threaded handlers have executed. Typically this callback masks level-triggered interrupts in an interrupt controller while leaving edge-triggered interrupts alone. Once all threaded handlers have executed, the *post_ithread* callback is invoked from the interrupt thread to enable future interrupts. Typically this callback unmasks level-triggered interrupts in an interrupt controller.

RETURN VALUES

The **intr_event_add_handler**(), **intr_event_create**(), **intr_event_destroy**(), **intr_event_handle**(), and **intr_event_remove_handler**() functions return zero on success and non-zero on failure. The **intr_priority**() function returns a process priority corresponding to the passed in interrupt flags.

EXAMPLES

The swi_add(9) function demonstrates the use of **intr_event_create**() and **intr_event_add_handler**().

```
int
swi_add(struct intr_event **eventp, const char *name, driver_intr_t handler,
    void *arg, int pri, enum intr_type flags, void **cookiep)
{
        struct intr_event *ie;
        int error = 0;

        if (flags & INTR_ENTROPY)
                return (EINVAL);

        ie = (eventp != NULL) ? *eventp : NULL;

        if (ie != NULL) {
                if (!(ie->ie_flags & IE_SOFT))
                        return (EINVAL);
        } else {
                error = intr_event_create(&ie, NULL, IE_SOFT, 0,
                    NULL, NULL, NULL, swi_assign_cpu, "swi%d:", pri);
                if (error)
                        return (error);
                if (eventp != NULL)
                        *eventp = ie;
        }
        if (handler != NULL) {
                error = intr_event_add_handler(ie, name, NULL, handler, arg,
                    PI_SWI(pri), flags, cookiep);
        }
```

```
        return (error);
}
```

ERRORS

The **intr_event_add_handler**() function will fail if:

[EINVAL] The *ie* or *name* arguments are NULL.

[EINVAL] The *handler* and *filter* arguments are both NULL.

[EINVAL] The IH_EXCLUSIVE flag is specified and the interrupt thread *ie* already has at least one handler, or the interrupt thread *ie* already has an exclusive handler.

The **intr_event_create**() function will fail if:

[EINVAL] A flag other than IE_SOFT was specified in the *flags* parameter.

The **intr_event_destroy**() function will fail if:

[EINVAL] The *ie* argument is NULL.

[EBUSY] The interrupt event pointed to by *ie* has at least one handler which has not been removed with **intr_event_remove_handler**().

The **intr_event_handle**() function will fail if:

[EINVAL] The *ie* argument is NULL.

[EINVAL] There are no interrupt handlers assigned to *ie*.

[EINVAL] The interrupt was not acknowledged by any filter and has no associated thread handler.

The **intr_event_remove_handler**() function will fail if:

[EINVAL] The *cookie* argument is NULL.

SEE ALSO

critical(9), kthread(9), locking(9), malloc(9), swi(9), uma(9)

HISTORY

Interrupt threads and their corresponding API first appeared in FreeBSD 5.0.