21.6 Spinlocks and Interrupt Events from ISRs

21.6.1 POSIX Spinlocks

Spinlocks, similarly to semaphores and mutexes, are mutual exclusion devices for protecting critical sections. The calling task blocks if it requests an unavailable semaphore or mutex. In contrast, the calling task of a spinlock will not block if the spinlock is not available; instead, it goes into a *tight loop* where it repeatedly checks the spinlock until it becomes available. This loop gives the "spin" part of a spinlock.

Table 21.3 lists the POSIX functions for spinlocks.

Note that the pthread_spin_lock operation checks and grabs a spinlock in an atomic manner. This ensures that only one spinning thread, even if there are several, can obtain the spinlock. Also, spinlocks are intended for use on preemptive systems. If a task in a nonpreemptive single-processor system ever went spinning on a lock, it would spin forever; no other thread would ever be able to obtain the CPU to release the lock.

Depending on the OS implementation, the above POSIX functions for spinlocks may or may not be used in ISRs.

21.6.2 QNX Event Structure

In addition to POSIX signals, QNX also supports other asynchronous notification mechanisms such as interrupts and pulses (see Section 21.7 for QNX pulses). These are treated in QNX as different types of events, and QNX has implemented a dedicated subsystem to uniformly handle different types of events.

An event object is described by a data structure named `sigevent`. Its members and the values each member can take are given in Table 21.4.

Table 21.3 POSIX.1 function calls for spinlocks

POSIX Function	Description
pthread_spin_destroy	Destroy a thread spinlock object
pthread_spin_init	Initialize a thread spinlock object
pthread_spin_lock	The calling thread acquires the lock if it is not held by another thread; otherwise, the thread spins
pthread_spin_trylock	The calling thread acquires the lock if it is not held by another thread; otherwise, the call returns with failure
pthread_spin_unlock	Release a thread spinlock. A thread spinning on the lock will acquire the lock

Table 21.4 QNX sigevent structure

sigenv_notify	sigev_signo	sigev_value	sigev_coid	sigev_priority	sigev_code
SIGEV_NONE					
SIGEV_SIGNAL	Signal				
SIGEV_THREAD		Value			
SIGEV_INTR					
SIGEV_PULSE		Value	Connection	Priority	Code
SIGEV_SIGNAL_CODE	Signal	Value			Code
SIGEV_SIGNAL_THREAD	Signal	Value			Code
SIGEV_UNBLOCK					

The value of sigenv_notify indicates the event type:

- SIGEV_NONE—a POSIX event indicating a null notification;
- SIGEV_SIGNAL—a POSIX signal event without code or a value;
- SIGEV_THREAD—a POSIX event to create a new thread (say, an OS timer can be set up to raise an event of this type to create a new thread every time it expires);
- SIGEV_INTR—a QNX event originated from a hardware interrupt (to be further handled at the user application level);
- SIGEV_PULSE—a QNX pulse event;
- SIGEV_SIGNAL_CODE—a QNX signal event with code and a value;
- SIGEV_SIGNAL_THREAD—a QNX signal event to a specific thread;
- SIGEV_UNBLOCK—a QNX event to force a thread to become unblocked.

Among the above event types, only the constants SIGEV_NONE, SIGEV_SIGNAL, and SIGEV_THREAD are defined in the POSIX.1-2008 standard. The others are QNX specific. Also, the POSIX `sigevent` structure defines only sigenv_notify, sigev_signo, and sigev_value; the other members are QNX specific.

QNX also provides several macros for initializing various types of events. For example, the macro SIGEV_SIGNAL_INIT(&evt, signo) is used to initialize a signal event *evt* with *signo* as the signal type to be raised. Given an event object *evt*, its type can be returned by calling the macro SIGEV_GET_TYPE(&evt).

21.6.3 Interrupt Handling in QNX Applications

Table 21.5 lists the QNX functions for interrupt handling at the user application level.

By calling the function InterruptAttach(), a user task (process or thread) can attach an interrupt handler (ISR) to a hardware interrupt source. The user task is also called the attaching process (or the attaching thread) of the interrupt handler installed via InterruptAttach(). Upon an interrupt from the hardware source, an interrupt handler can

Table 21.5 QNX function calls for interrupt handling

POSIX Function	Description
InterruptAttach	Attach an interrupt handler to a hardware interrupt source
InterruptDetach	Detach an interrupt handler by ID
InterruptDisable	Disable all hardware interrupts
InterruptEnable	Enable all hardware interrupts
InterruptLock	Guard a critical section by a spinlock object shared between an interrupt handler and a thread. This call disables interrupts, spinning in a tight loop until the spinlock is acquired
InterruptMask	Disable a specific hardware interrupt
InterruptUnlock	Unlock the specified spinlock and re-enable interrupts
InterruptUnmask	Enable a specific hardware interrupt
InterruptWait	The calling thread blocks, waiting for the interrupt handler it attached before to return a hardware interrupt event of type SIGEV_INTR

(selectively) deliver a SIGEV_INTR event to the attaching process (or thread). In such a sense, the attaching process (or thread) of an interrupt handler actually acts as the server task of the handler. The interrupt handler is removed when the attaching process or thread exits.

When a handler is attached, by default it is placed in front of any existing handlers for that interrupt and is called first (see Section 4.5.3 for ISR chaining). By the setting of a flag, an interrupt handler can also be placed at the end of any existing handlers.

The QNX implementation of pthread_spin_lock and pthread_spin_unlock cannot be used in ISRs. Instead, the functions InterruptLock() and InterruptUnLock() allow both ISRs and threads to use a spinlock. The calling task of InterruptLock() tries to acquire a spinlock. If the lock is not immediately available, the task spins in a tight loop until the lock is acquired. It can be used to protect access to shared data structures between an interrupt handler and the attaching thread of the handler.

21.6.4 Example: Interrupt Events from ISRs

Figure 21.3 gives a design showing the use of spinlock objects. A QNX-based implementation of this design is given in Listing 21.6. This example helps us to learn the following points:

(1) The program first performs initialization (lines 233-264):
   (a) It calls setupTerminal() to disable echo of the keyboard inputs.
   (b) It attaches a handler for terminal interrupt signals (SIGINT). Once the program has been interrupted by Ctrl-C, the signal handler sig_handler() is responsible for restoring the system settings changed by this program.
   (c) It initializes the two character arrays *time_comm_ptr* and *key_comm_ptr*.
   (d) It initializes two event objects of type SIGEV_INTR: *keyevent* and *timeevent*.
   (e) It creates a semaphore object referenced by a pointer: *sem*.

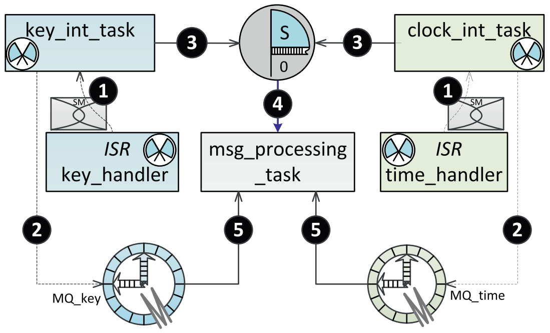

Figure 21.3
Interrupt locks and interrupt events.

(f) It opens and initializes two message queues: *MQ_key* and *MQ_time*. Both have a capacity of 20 and enable nonblocking access.

(g) Lastly, it initializes two spinlock objects referenced by the pointers *klock* and *tlock*.

(2) In addition to the main thread, the process creates three task threads:

(a) msg_processing_task—for message processing;

(b) key_int_task—for servicing events resulted from the x86 keyboard interrupts; and

(c) clock_int_task—for servicing events resulted from the x86 real-time clock interrupts (say, raised every 1 ms).

The two interrupt service tasks have a higher priority than the message processing task.

(3) The keyboard ISR, defined by the function key_handler(), is attached by the corresponding service task key_int_task. Likewise, the clock ISR, defined by the function time_handler(), is attached by the clock_int_task.

(4) The communication area *key_comm_ptr*, denoted by the "shared memory object" in Figure 21.3, is shared by both key_handler() and its service task key_int_task(). Hence, their critical sections are protected by the spinlock object *klock* via the function calls InterruptLock() and InterruptUnlock(), respectively. Note that it is not necessary to use a spinlock object when no data are shared between an ISR and its service task.

(5) Similarly, the communication area *time_comm_ptr* is shared by both time_handler() and its service task clock_int_task(). Their critical sections are protected by the spinlock object *tlock*.

(6) The keyboard ISR (key_handler) returns a keyevent to its service task every other time that the keyboard has raised interrupts. Since a hardware interrupt is raised for both a key-down event and a key-up event, the overall effect of this keyboard ISR is to ensure that the service task is notified only once for each key press (a key-down event followed by a key-up event). Similarly, the clock ISR (time_handler) returns a timeevent to its service task every 1000 times that the clock has raised interrupts. If the real-time clock raises an interrupt every 1 ms, then the overall effect of this clock ISR is to ensure that the service task is notified each second.