torvalds / linux Public

Code Pull requests 312 Actions Projects Security Insights

master

linux / include / linux / spinlock.h

Sebastian Andrzej Siewior locking: Detect includes rwlock.h outside of spinlock.h ... History

51 contributors +18

497 lines (426 sloc) | 13.8 KB

```
/* SPDX-License-Identifier: GPL-2.0 */
#ifndef __LINUX_SPINLOCK_H
#define __LINUX_SPINLOCK_H
#define __LINUX_INSIDE_SPINLOCK_H

/*
 * include/linux/spinlock.h - generic spinlock/rwlock declarations
 *
 * here's the role of the various spinlock/rwlock related include files:
 *
 * on SMP builds:
 *
 *  asm/spinlock_types.h: contains the arch_spinlock_t/arch_rwlock_t and the
 *                        initializers
 *
 *  linux/spinlock_types_raw:
 *                        The raw types and initializers
 *  linux/spinlock_types.h:
 *                        defines the generic type and initializers
 *
 *  asm/spinlock.h:       contains the arch_spin_*()/etc. lowlevel
 *                        implementations, mostly inline assembly code
 *
 *   (also included on UP-debug builds:)
 *
 *  linux/spinlock_api_smp.h:
 *                        contains the prototypes for the _spin_*() APIs.
 *
 *  linux/spinlock.h:     builds the final spin_*() APIs.
 *
 * on UP builds:
 *
 *  linux/spinlock_type_up.h:
 *                        contains the generic, simplified UP spinlock type.
 *                        (which is an empty structure on non-debug builds)
 *
 *  linux/spinlock_types_raw:
 *                        The raw RT types and initializers
 *  linux/spinlock_types.h:
 *                        defines the generic type and initializers
 *
 *  linux/spinlock_up.h:
 *                        contains the arch_spin_*()/etc. version of UP
 *                        builds. (which are NOPs on non-debug, non-preempt
 *                        builds)
 *
 *   (included on UP-non-debug builds:)
 *
 *  linux/spinlock_api_up.h:
 *                        builds the _spin_*() APIs.
 *
 *  linux/spinlock.h:     builds the final spin_*() APIs.
 */

#include <linux/typecheck.h>
#include <linux/preempt.h>
#include <linux/linkage.h>
#include <linux/compiler.h>
#include <linux/irqflags.h>
#include <linux/thread_info.h>
#include <linux/stringify.h>
#include <linux/bottom_half.h>
#include <linux/lockdep.h>
#include <asm/barrier.h>
#include <asm/mmiowb.h>


/*
 * Must define these before including other files, inline functions need them
 */
#define LOCK_SECTION_NAME ".text..lock."KBUILD_BASENAME

#define LOCK_SECTION_START(extra)               \
        ".subsection 1\n\t"                     \
        extra                                   \
        ".ifndef " LOCK_SECTION_NAME "\n\t"     \
        LOCK_SECTION_NAME ":\n\t"               \
        ".endif\n"
```

```

#define LOCK_SECTION_END                        \
        ".previous\n\t"

#define __lockfunc __section(".spinlock.text")

/*
 * Pull the arch_spinlock_t and arch_rwlock_t definitions:
 */
#include <linux/spinlock_types.h>

/*
 * Pull the arch_spin*() functions/declarations (UP-nondebug doesn't need them):
 */
#ifdef CONFIG_SMP
# include <asm/spinlock.h>
#else
# include <linux/spinlock_up.h>
#endif

#ifdef CONFIG_DEBUG_SPINLOCK
  extern void __raw_spin_lock_init(raw_spinlock_t *lock, const char *name,
                                   struct lock_class_key *key, short inner);

# define raw_spin_lock_init(lock)                                       \
do {                                                                    \
        static struct lock_class_key __key;                             \
                                                                        \
        __raw_spin_lock_init((lock), #lock, &__key, LD_WAIT_SPIN);      \
} while (0)

#else
# define raw_spin_lock_init(lock)                               \
        do { *(lock) = __RAW_SPIN_LOCK_UNLOCKED(lock); } while (0)
#endif

#define raw_spin_is_locked(lock)        arch_spin_is_locked(&(lock)->raw_lock)

#ifdef arch_spin_is_contended
#define raw_spin_is_contended(lock)     arch_spin_is_contended(&(lock)->raw_lock)
#else
#define raw_spin_is_contended(lock)     (((void)(lock), 0))
#endif /*arch_spin_is_contended*/

/*
 * smp_mb__after_spinlock() provides the equivalent of a full memory barrier
 * between program-order earlier lock acquisitions and program-order later
 * memory accesses.
 *
 * This guarantees that the following two properties hold:
 *
 *   1) Given the snippet:
 *
 *        { X = 0;  Y = 0; }
 *
 *        CPU0                          CPU1
 *
 *        WRITE_ONCE(X, 1);             WRITE_ONCE(Y, 1);
 *        spin_lock(S);                 smp_mb();
 *        smp_mb__after_spinlock();     r1 = READ_ONCE(X);
 *        r0 = READ_ONCE(Y);
 *        spin_unlock(S);
 *
 *      it is forbidden that CPU0 does not observe CPU1's store to Y (r0 = 0)
 *      and CPU1 does not observe CPU0's store to X (r1 = 0); see the comments
 *      preceding the call to smp_mb__after_spinlock() in __schedule() and in
 *      try_to_wake_up().
 *
 *   2) Given the snippet:
 *
 *  { X = 0;  Y = 0; }
 *
 *  CPU0                CPU1                            CPU2
 *
 *  spin_lock(S);       spin_lock(S);                   r1 = READ_ONCE(Y);
 *  WRITE_ONCE(X, 1);   smp_mb__after_spinlock();       smp_rmb();
 *  spin_unlock(S);     r0 = READ_ONCE(X);              r2 = READ_ONCE(X);
 *                      WRITE_ONCE(Y, 1);
 *                      spin_unlock(S);
 *
 *      it is forbidden that CPU0's critical section executes before CPU1's
 *      critical section (r0 = 1), CPU2 observes CPU1's store to Y (r1 = 1)
 *      and CPU2 does not observe CPU0's store to X (r2 = 0); see the comments
 *      preceding the calls to smp_rmb() in try_to_wake_up() for similar
 *      snippets but "projected" onto two CPUs.
 *
 * Property (2) upgrades the lock to an RCsc lock.
 *
 * Since most load-store architectures implement ACQUIRE with an smp_mb() after
 * the LL/SC loop, they need no further barriers. Similarly all our TSO
 * architectures imply an smp_mb() for each atomic instruction and equally don't
 * need more.
 *
 * Architectures that can implement ACQUIRE better need to take care.
 */
#ifndef smp_mb__after_spinlock
#define smp_mb__after_spinlock()        kcsan_mb()
#endif
```

```

#ifdef CONFIG_DEBUG_SPINLOCK
 extern void do_raw_spin_lock(raw_spinlock_t *lock) __acquires(lock);
 extern int do_raw_spin_trylock(raw_spinlock_t *lock);
 extern void do_raw_spin_unlock(raw_spinlock_t *lock) __releases(lock);
#else
static inline void do_raw_spin_lock(raw_spinlock_t *lock) __acquires(lock)
{
        __acquire(lock);
        arch_spin_lock(&lock->raw_lock);
        mmiowb_spin_lock();
}

static inline int do_raw_spin_trylock(raw_spinlock_t *lock)
{
        int ret = arch_spin_trylock(&(lock)->raw_lock);

        if (ret)
                mmiowb_spin_lock();

        return ret;
}

static inline void do_raw_spin_unlock(raw_spinlock_t *lock) __releases(lock)
{
        mmiowb_spin_unlock();
        arch_spin_unlock(&lock->raw_lock);
        __release(lock);
}
#endif

/*
 * Define the various spin_lock methods.  Note we define these
 * regardless of whether CONFIG_SMP or CONFIG_PREEMPTION are set. The
 * various methods are defined as nops in the case they are not
 * required.
 */
#define raw_spin_trylock(lock)  __cond_lock(lock, _raw_spin_trylock(lock))

#define raw_spin_lock(lock)     _raw_spin_lock(lock)

#ifdef CONFIG_DEBUG_LOCK_ALLOC
# define raw_spin_lock_nested(lock, subclass) \
        _raw_spin_lock_nested(lock, subclass)

# define raw_spin_lock_nest_lock(lock, nest_lock)                      \
         do {                                                          \
                typecheck(struct lockdep_map *, &(nest_lock)->dep_map);\
                _raw_spin_lock_nest_lock(lock, &(nest_lock)->dep_map); \
         } while (0)
#else
/*
 * Always evaluate the 'subclass' argument to avoid that the compiler
 * warns about set-but-not-used variables when building with
 * CONFIG_DEBUG_LOCK_ALLOC=n and with W=1.
 */
# define raw_spin_lock_nested(lock, subclass)           \
        _raw_spin_lock(((void)(subclass), (lock)))
# define raw_spin_lock_nest_lock(lock, nest_lock)       _raw_spin_lock(lock)
#endif

#if defined(CONFIG_SMP) || defined(CONFIG_DEBUG_SPINLOCK)

#define raw_spin_lock_irqsave(lock, flags)                      \
        do {                                            \
                typecheck(unsigned long, flags);        \
                flags = _raw_spin_lock_irqsave(lock);   \
        } while (0)

#ifdef CONFIG_DEBUG_LOCK_ALLOC
#define raw_spin_lock_irqsave_nested(lock, flags, subclass)             \
        do {                                                            \
                typecheck(unsigned long, flags);                        \
                flags = _raw_spin_lock_irqsave_nested(lock, subclass);  \
        } while (0)
#else
#define raw_spin_lock_irqsave_nested(lock, flags, subclass)             \
        do {                                                            \
                typecheck(unsigned long, flags);                        \
                flags = _raw_spin_lock_irqsave(lock);                   \
        } while (0)
#endif

#else

#define raw_spin_lock_irqsave(lock, flags)              \
        do {                                            \
                typecheck(unsigned long, flags);        \
                _raw_spin_lock_irqsave(lock, flags);    \
        } while (0)

#define raw_spin_lock_irqsave_nested(lock, flags, subclass)     \
        raw_spin_lock_irqsave(lock, flags)

#endif

#define raw_spin_lock_irq(lock)         _raw_spin_lock_irq(lock)
#define raw_spin_lock_bh(lock)          _raw_spin_lock_bh(lock)
```

```
#define raw_spin_unlock(lock)           _raw_spin_unlock(lock)
#define raw_spin_unlock_irq(lock)       _raw_spin_unlock_irq(lock)

#define raw_spin_unlock_irqrestore(lock, flags)         \
        do {                                            \
                typecheck(unsigned long, flags);        \
                _raw_spin_unlock_irqrestore(lock, flags);       \
        } while (0)
#define raw_spin_unlock_bh(lock)        _raw_spin_unlock_bh(lock)

#define raw_spin_trylock_bh(lock) \
        __cond_lock(lock, _raw_spin_trylock_bh(lock))

#define raw_spin_trylock_irq(lock) \
({ \
        local_irq_disable(); \
        raw_spin_trylock(lock) ? \
        1 : ({ local_irq_enable(); 0;  }); \
})

#define raw_spin_trylock_irqsave(lock, flags) \
({ \
        local_irq_save(flags); \
        raw_spin_trylock(lock) ? \
        1 : ({ local_irq_restore(flags); 0; }); \
})

#ifndef CONFIG_PREEMPT_RT
/* Include rwlock functions for !RT */
#include <linux/rwlock.h>
#endif

/*
 * Pull the _spin_*()/_read_*()/_write_*() functions/declarations:
 */
#if defined(CONFIG_SMP) || defined(CONFIG_DEBUG_SPINLOCK)
# include <linux/spinlock_api_smp.h>
#else
# include <linux/spinlock_api_up.h>
#endif

/* Non PREEMPT_RT kernel, map to raw spinlocks: */
#ifndef CONFIG_PREEMPT_RT

/*
 * Map the spin_lock functions to the raw variants for PREEMPT_RT=n
 */

static __always_inline raw_spinlock_t *spinlock_check(spinlock_t *lock)
```

```
{
        return &lock->rlock;
}

#ifdef CONFIG_DEBUG_SPINLOCK

# define spin_lock_init(lock)                                   \
do {                                                            \
        static struct lock_class_key __key;                     \
                                                                \
        __raw_spin_lock_init(spinlock_check(lock),              \
                             #lock, &__key, LD_WAIT_CONFIG);    \
} while (0)

#else

# define spin_lock_init(_lock)                  \
do {                                            \
        spinlock_check(_lock);                  \
        *(_lock) = __SPIN_LOCK_UNLOCKED(_lock); \
} while (0)

#endif

static __always_inline void spin_lock(spinlock_t *lock)
{
        raw_spin_lock(&lock->rlock);
}

static __always_inline void spin_lock_bh(spinlock_t *lock)
{
        raw_spin_lock_bh(&lock->rlock);
}

static __always_inline int spin_trylock(spinlock_t *lock)
{
        return raw_spin_trylock(&lock->rlock);
}

#define spin_lock_nested(lock, subclass)                        \
do {                                                            \
        raw_spin_lock_nested(spinlock_check(lock), subclass);   \
} while (0)

#define spin_lock_nest_lock(lock, nest_lock)                            \
do {                                                                    \
        raw_spin_lock_nest_lock(spinlock_check(lock), nest_lock);       \
} while (0)

```

```
static __always_inline void spin_lock_irq(spinlock_t *lock)
{
	raw_spin_lock_irq(&lock->rlock);
}

#define spin_lock_irqsave(lock, flags)				\
do {								\
	raw_spin_lock_irqsave(spinlock_check(lock), flags);	\
} while (0)

#define spin_lock_irqsave_nested(lock, flags, subclass)			\
do {									\
	raw_spin_lock_irqsave_nested(spinlock_check(lock), flags, subclass); \
} while (0)

static __always_inline void spin_unlock(spinlock_t *lock)
{
	raw_spin_unlock(&lock->rlock);
}

static __always_inline void spin_unlock_bh(spinlock_t *lock)
{
	raw_spin_unlock_bh(&lock->rlock);
}

static __always_inline void spin_unlock_irq(spinlock_t *lock)
{
	raw_spin_unlock_irq(&lock->rlock);
}

static __always_inline void spin_unlock_irqrestore(spinlock_t *lock, unsigned long flags)
{
	raw_spin_unlock_irqrestore(&lock->rlock, flags);
}

static __always_inline int spin_trylock_bh(spinlock_t *lock)
{
	return raw_spin_trylock_bh(&lock->rlock);
}

static __always_inline int spin_trylock_irq(spinlock_t *lock)
{
	return raw_spin_trylock_irq(&lock->rlock);
}

#define spin_trylock_irqsave(lock, flags)			\
({								\
	raw_spin_trylock_irqsave(spinlock_check(lock), flags); \
})
```

```

/**
 * spin_is_locked() - Check whether a spinlock is locked.
 * @lock: Pointer to the spinlock.
 *
 * This function is NOT required to provide any memory ordering
 * guarantees; it could be used for debugging purposes or, when
 * additional synchronization is needed, accompanied with other
 * constructs (memory barriers) enforcing the synchronization.
 *
 * Returns: 1 if @lock is locked, 0 otherwise.
 *
 * Note that the function only tells you that the spinlock is
 * seen to be locked, not that it is locked on your CPU.
 *
 * Further, on CONFIG_SMP=n builds with CONFIG_DEBUG_SPINLOCK=n,
 * the return value is always 0 (see include/linux/spinlock_up.h).
 * Therefore you should not rely heavily on the return value.
 */
static __always_inline int spin_is_locked(spinlock_t *lock)
{
        return raw_spin_is_locked(&lock->rlock);
}

static __always_inline int spin_is_contended(spinlock_t *lock)
{
        return raw_spin_is_contended(&lock->rlock);
}

#define assert_spin_locked(lock)        assert_raw_spin_locked(&(lock)->rlock)

#else  /* !CONFIG_PREEMPT_RT */
# include <linux/spinlock_rt.h>
#endif /* CONFIG_PREEMPT_RT */

/*
 * Pull the atomic_t declaration:
 * (asm-mips/atomic.h needs above definitions)
 */
#include <linux/atomic.h>
/**
 * atomic_dec_and_lock - lock on reaching reference count zero
 * @atomic: the atomic counter
 * @lock: the spinlock in question
 *
 * Decrements @atomic by 1.  If the result is 0, returns true and locks
 * @lock.  Returns false for all other cases.
 */
extern int _atomic_dec_and_lock(atomic_t *atomic, spinlock_t *lock);
```

```
#define atomic_dec_and_lock(atomic, lock) \
                __cond_lock(lock, _atomic_dec_and_lock(atomic, lock))

extern int _atomic_dec_and_lock_irqsave(atomic_t *atomic, spinlock_t *lock,
                                        unsigned long *flags);
#define atomic_dec_and_lock_irqsave(atomic, lock, flags) \
                __cond_lock(lock, _atomic_dec_and_lock_irqsave(atomic, lock, &(flags)))

int __alloc_bucket_spinlocks(spinlock_t **locks, unsigned int *lock_mask,
                             size_t max_size, unsigned int cpu_mult,
                             gfp_t gfp, const char *name,
                             struct lock_class_key *key);

#define alloc_bucket_spinlocks(locks, lock_mask, max_size, cpu_mult, gfp)    \
        ({                                                                   \
                static struct lock_class_key key;                            \
                int ret;                                                     \
                                                                             \
                ret = __alloc_bucket_spinlocks(locks, lock_mask, max_size,   \
                                               cpu_mult, gfp, #locks, &key); \
                ret;                                                         \
        })

void free_bucket_spinlocks(spinlock_t *locks);

#undef __LINUX_INSIDE_SPINLOCK_H
#endif /* __LINUX_SPINLOCK_H */
```